C MAGAZINE

改訂第2版

ゲームプログラミング

遊びのレシピ

アルゴリズムとデータ構造

C/C++ & Delphi 対応

有馬元嗣／著

SOFT BANK Publishing

C MAGAZINE

改訂第2版

ゲームプログラミング

遊びのレシピ

アルゴリズムとデータ構造

有馬元嗣／著

SOFT BANK Publishing

C MAGAZINE

改訂第2版

ゲームプログラミング

遊びのレシピ

アルゴリズムとデータ構造

有馬元嗣／著

SOFT BANK Publishing

Borland DelphiおよびBorland C++Builderはボーランド(株)の商品です。
本文中の商品名は，一般に各社の登録商標です。
本文中に，TM，®マークは明記しておりません。

はじめに

ある夏の日に「そういえばシューティングゲームのアルゴリズムってどんなのだっけ？」と思ったところから，この本は始まりました。先輩諸氏のゲームプログラミングに関する本を読んでもハードウェア/ソフトウェアの利用ガイドとしか思えないものばかりでした。「なければ自分で作る」主義の筆者は，ゲームを作るうえで要所や定番となるアルゴリズムとデータ構造をあちこちから集め，現在は休刊となっている『Inside Windows』誌（ソフトバンク パブリッシング刊）にて「遊びのレシピ」として発表させていただきました（創刊1号～1997年5月号）。その後，この内容をさらに加筆修正して書籍となった『遊びのレシピ』が刊行されました。

あれから4年……。おかげさまで『遊びのレシピ』は多くの方々に読んでいただいたようです。その間に古い記述も目立つようになりました。そこでサンプルプログラムをC++Builder 5，Delphi 5に対応させ，いくつかの記述を修正しました。もちろん，ゲームプログラミングの仕組みやその楽しさについて解説するという内容は変わりません。とくに「ゲームを作りたいけどどこから手をつけたらいいの？」，「個々の処理はわかるけど，どのように繋げていけばいいの？」といった方に読んでいただければと思います。

さらにC MAGAZINE編集部のご厚意により，WonderWitch上のゲームを作成しながらゲームの仕組みを解き明かす連載「遊びのレシピ for WonderWitch」の誌面やファイルをCD-ROMに収録させていただきました。WonderWitchを使ったゲームプログラミングの参考になれば幸いです。

この本に掲載されたグラフィックデータは，制作したMALUMI氏，WonderWitch版は迎夢達人氏に著作権があります。ソースコードは筆者に著作権がありますが，改変，配布は自由に行ってください。ただし掲載されたままの状態のファイルは許可なく配布しないでください。ご質問やご要望は，http://cmaga.zdnet.co.jp/books/recipe/で受け付けています。

最後にこの本を執筆するにあたり，制作にかかわっていただいた方々，いつのまにやら職場となってしまったC MAGAZINE編集部のみなさん，WonderWitch関連でご迷惑をおかけしましたキュート様やバンダイ様，漫画家のMALUMI氏，新谷勇雄氏，いつも会社から帰れなくてどこにも遠出できず迷惑をかけている妻の綾に心から感謝します。

この本をきっかけにプログラミングの楽しさに気づいた方が1人でも増えたらとてもうれしく思います。それではゲームを作るレシピをゆっくりとご堪能ください。

2001年5月

有馬元嗣

CONTENTS

Section4　シミュレーションゲーム

Section5　シューティングゲーム

Section6　カードゲーム

Section7　麻雀ゲーム

Chapter 1

ゲームの仕組みを知ろう

この章では，この本の目的とゲームプログラミングでよく使われている「手法」について取りあげています。このあとの章で述べられているアルゴリズムへ触れる前に，ひととおりこの章を読んでおけば，理解の助けとなることでしょう。

Section 1 ゲームのアルゴリズムとは？

ゲームを作るうえで必要不可欠な知識とはどういったものでしょうか。プログラムに使用する言語や開発環境についての知識は当然必要でしょう。しかし，「本当に自分だけのゲーム」を作るためには，それだけでは十分ではありません。そのゲームはどのように動いているのか，つまりゲームの「仕組み」を知っておく必要があるのです。

● アルゴリズム＝「ゲームの仕組み」

物事が動いている「仕組み」というのは，とてもおもしろいものです。とくにそれがふだん目にすることができないものであれば，なおさら興味を引かれます。国家の運営などもそうでしょうし，飛行機の離着陸，会社の経営などもそうです。これまでは，ある特定の人間にしか実際に行うことができなかったことが，現在ではほとんどシミュレーションという形でゲームになり，誰もが擬似的に楽しめるようになりました。

では「ゲームの仕組み」とは何でしょう？　あの鮮やかな絵や効果音が鳴り響き，興奮と感動を与えてくれるゲームとは，どうやって作られているのでしょうか？

おおざっぱにいえば，この「ゲームを動かすための仕組み」のうち，ゲームソフトを作る「プログラミング」作業での「作り方」「考え方」こそが「アルゴリズム」ということになります。

◆プログラミングのための考え方

それでは，アルゴリズムとは具体的にはどのようなものなのでしょうか？　格闘ゲームを例に取ると，「CPU側のキャラクタを動かすための思考ルーチン」「背景/キャラクタなどの画面への表示」「プレイヤーがキー入力した技の処理」「得点や当たり判定などのゲーム進行」，これら全体を統括して動かす「ゲーム全体の流れ」がそれぞれのアルゴリズムということになります。

表示関係では，タイマやグラフィック処理など，ゲームを作る環境によってプログラミングの内容はまったく変わってきますが，「タイマを使ってキャラクタの絵を変える」「キャラクタの絵は何層にも重ねるようにして管理する」などという，

直接環境とは関係のない「プログラミングするための考え方」がとても重要です。

ゲーム業界では「麻雀/将棋などのアルゴリズムを持ったソフトハウスはつぶれない」という話があります。これはどんな新機種が出ても，そのゲームの本質となるアルゴリズムさえあれば，いちはやく製品を市場に投入できるからです。

すべてのゲームには，こうした基本となるアルゴリズムがあります。「RPG」「アドベンチャー」「格闘ゲーム」などすべてそうです。これはパソコン，コンシューマ，アーケードとマシンが違ってもまったく同じです。表示方法やキャラクタなどの外見は変わりますが，どんなに新しいゲームでもアルゴリズムそのものはたいして変わりません。そして各ソフトハウスは，こうしたアルゴリズムを作るのに知恵と工夫を凝らしています。あるソフトハウスにしか存在しない，門外不出のアルゴリズムというものもよく聞かれます。

◆本書の内容と目的

いままでゲームを取り扱った書籍は「ゲームを通じてプログラミングを習う」といった趣旨の内容がほとんどでした。たしかに，これだけでもゲームを作ることはできますが，根本となる仕組みを理解していないと「本当に自分だけのゲーム」を作りあげることはできません。

この本では「プログラミングそのもの」だけではなく，より高い視点から見た「ゲームはどういうアルゴリズムとデータ構造で動いているのか」ということを主題にしています。もちろん，それらに関連するプログラミングの方法もあわせて述べていきます。内容的にやや難しいところが出てくるかもしれませんが，「こんなゲームではこんなことをしている」といった「アルゴリズムのおもしろさ」をぜひ楽しんでください。

● 使用する言語と開発環境

従来のプログラミング関係書籍は「ある特定の言語」を紹介しているものがほとんどです。読者の側からしてみれば，実際に自分が使う処理系に対応していなければ，本を買う意味がないかもしれません。でもアルゴリズムとは言語の枠を越えた「実現するためのもっともよい考え方」です。「特定の言語」に縛られると，アルゴリズムの楽しさを多くの人に知ってもらうことができなくなってしまいます。

◆C++とObjectPascalを使用

本書でアルゴリズムの解説に使う言語は「ObjectPascal」,「CまたはC++」の2つをあわせて掲載するようにしました。アルゴリズムそのものは，どの言語でも使えるものばかりなので，ほかの言語を使いたい人も簡単に利用できると思います。まずは動く仕組みを知ってください。

でも，それだけではおもしろくありません。せっかく仕組みがわかってもすぐに実践できなければ，ただの空論に終わってしまいます。そこで，現在主流となっているWindowsの環境でゲーム作成ができるように，Borland社から発売されている「C++Builder」,「Delphi」で作成したサンプルプログラムを掲載しました。本書では，C++Builderで作成したプログラムは「C/C++」, Delphiで作成したプログラムは「Delphi」とList名に表記します。

サンプルプログラムのなかには，このあとに解説を行うゲーム開発に必要な「スプライト」,「スクロール」などに関係するコンポーネントも含まれています。本書の後半では，これら2つの処理系に関係するプログラミングについて詳しく取りあげてあるので，参考にしてみてください。

サンプルプログラムは，Win32で動作するプログラムなので，Windows 95/98/MeやWindows NT/2000などで遊ぶことができます。

Section 2 ゲームプログラミングのテクニック

ゲームのアルゴリズムに触れる前に，ゲームでよく使われるプログラミングテクニックについて解説します。こうした方法を知っておくと，その方法に合わせてアルゴリズムを考えることができます。発想が逆のようですが，実際にはそのほうが実用的です。本章で述べる方法は，あとの章にも登場します。Chapter2以降で基本的な方法がわからなくなったら，本章で復習してください。
それではいくつかのテクニックを見ていきましょう。

テーブル参照

ミステリ小説を読んでいるとき，知り合いから「ああ，その本ね。犯人は○×でトリックはこうなんだよ……」と解説されてしまったことはありませんか？「人がせっかく楽しんでいるのに」と，知り合いをうらんだところでどうにもなりません。とたんに読む気をなくしてしまいます。でも，そこでちょっと視点を変えて，「犯人を早く知りたいという目的で本を読んでいた」と考えたらどうでしょうか？　もし何かの答えを知りたいとき，あらかじめその答えがわかっていれば，もっとも短い時間で答えを得ることができます。逆説的ですがいわれてみれば当たり前のことです。

この考え方はプログラミングでも利用することができます。たとえば，次のような場面を考えてみましょう。

ある値が0～255の範囲にあり，これをある計算式で得られる値に変換したいとします。これはList 1-1のような関数になります。引数が「変換したい値」で返り値が「変換された結果」です。この例では255を与えると1にするといった，値を反転する動作をします。これを高速化したいときは，「値の範囲分だけ配列を用意」してやり，そこへ「あらかじめ計算しておいた値を入れて」おきます。実際に変換するときは，この配列から「対応する値を取り出す」ことをします(Fig. 1-1)。これを基にList 1-1を書き直したのがList 1-2です。C/C++ならList 1-3のようになります。

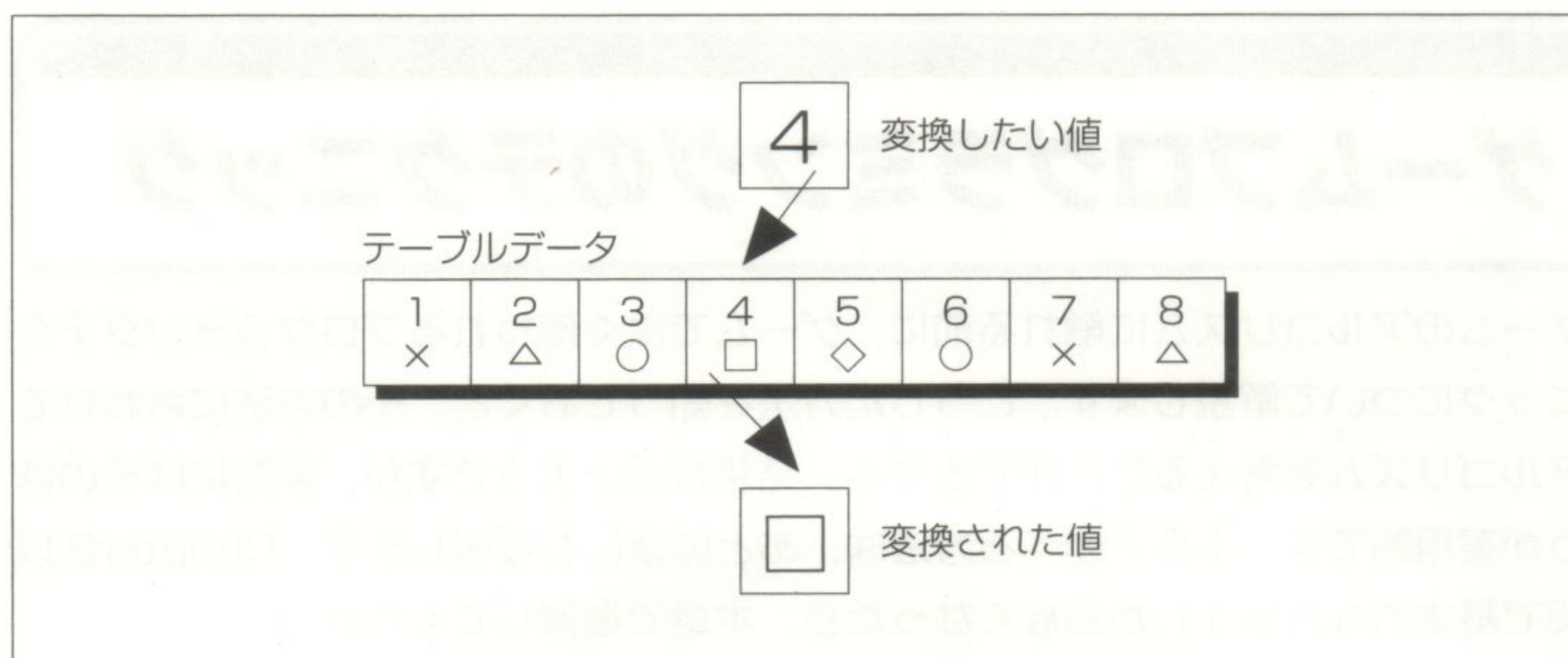



Fig. 1-1 ●必要な範囲分だけ配列を用意して，そこへあらかじめ計算しておいた値を入れておき，この配列から対応する値を取り出す

List 1-1 ●0～255の範囲の値を計算で反転する(Delphi)

```
{ 0～255の範囲の値を計算で反転するファンクション関数 }
{ (例) 1 なら返り値は 255, 0 なら 0                 }

function Exchange(i : integer) : integer;
begin
   if i = 0 then
      Exchange := 0
   else
      Exchange := i xor 255 + 1;
end;
```

List 1-2 ●配列を用意してList 1-1を書き直した(Delphi)

```
{ テーブルの例 }
{ 0 ～ 255 の範囲の値をテーブルで反転するファンクション関数 }
{ (例) 1 なら返り値は 255, 0 なら 0                       }

{ 定数宣言 }
const
   tbl : array[0..255] of integer = (
      0,    255,  254,  253,  252,  251,  250,  249,
      248,  247,  246,  245,  244,  243,  242,  241,
      240,  239,  238,  237,  236,  235,  234,  233,
      232,  231,  230,  229,  228,  227,  226,  225,
      224,  223,  222,  221,  220,  219,  218,  217,
      216,  215,  214,  213,  212,  211,  210,  209,
      208,  207,  206,  205,  204,  203,  202,  201,
      200,  199,  198,  197,  196,  195,  194,  193,
```

```
192,     191,     190,     189,     188,     187,     186,     185,
184,     183,     182,     181,     180,     179,     178,     177,
176,     175,     174,     173,     172,     171,     170,     169,
168,     167,     166,     165,     164,     163,     162,     161,
160,     159,     158,     157,     156,     155,     154,     153,
152,     151,     150,     149,     148,     147,     146,     145,
144,     143,     142,     141,     140,     139,     138,     137,
136,     135,     134,     133,     132,     131,     130,     129,
128,     127,     126,     125,     124,     123,     122,     121,
120,     119,     118,     117,     116,     115,     114,     113,
112,     111,     110,     109,     108,     107,     106,     105,
104,     103,     102,     101,     100,     99,      98,      97,
96,      95,      94,      93,      92,      91,      90,      89,
88,      87,      86,      85,      84,      83,      82,      81,
80,      79,      78,      77,      76,      75,      74,      73,
72,      71,      70,      69,      68,      67,      66,      65,
64,      63,      62,      61,      60,      59,      58,      57,
56,      55,      54,      53,      52,      51,      50,      49,
48,      47,      46,      45,      44,      43,      42,      41,
40,      39,      38,      37,      36,      35,      34,      33,
32,      31,      30,      29,      28,      27,      26,      25,
24,      23,      22,      21,      20,      19,      18,      17,
16,      15,      14,      13,      12,      11,      10,      9,
8,       7,       6,       5,       4,       3,       2,       1
);


{ 関数本体 }
function Exchange(i : integer) : integer;
begin
   Exchange := tbl[i];
end;
```

List 1-3●配列を利用して0～255の範囲の値を反転する(C/C++)

```
/* テーブルを利用した変換関数                    */

static int tbl[] = {
    0,       255,     254,     253,     252,     251,     250,     249,
    248,     247,     246,     245,     244,     243,     242,     241,
    240,     239,     238,     237,     236,     235,     234,     233,
    232,     231,     230,     229,     228,     227,     226,     225,
    224,     223,     222,     221,     220,     219,     218,     217,
    216,     215,     214,     213,     212,     211,     210,     209,
    208,     207,     206,     205,     204,     203,     202,     201,
    200,     199,     198,     197,     196,     195,     194,     193,
    192,     191,     190,     189,     188,     187,     186,     185,
    184,     183,     182,     181,     180,     179,     178,     177,
```

List 1-3

```
        176,    175,    174,    173,    172,    171,    170,    169,
        168,    167,    166,    165,    164,    163,    162,    161,
        160,    159,    158,    157,    156,    155,    154,    153,
        152,    151,    150,    149,    148,    147,    146,    145,
        144,    143,    142,    141,    140,    139,    138,    137,
        136,    135,    134,    133,    132,    131,    130,    129,
        128,    127,    126,    125,    124,    123,    122,    121,
        120,    119,    118,    117,    116,    115,    114,    113,
        112,    111,    110,    109,    108,    107,    106,    105,
        104,    103,    102,    101,    100,    99,     98,     97,
        96,     95,     94,     93,     92,     91,     90,     89,
        88,     87,     86,     85,     84,     83,     82,     81,
        80,     79,     78,     77,     76,     75,     74,     73,
        72,     71,     70,     69,     68,     67,     66,     65,
        64,     63,     62,     61,     60,     59,     58,     57,
        56,     55,     54,     53,     52,     51,     50,     49,
        48,     47,     46,     45,     44,     43,     42,     41,
        40,     39,     38,     37,     36,     35,     34,     33,
        32,     31,     30,     29,     28,     27,     26,     25,
        24,     23,     22,     21,     20,     19,     18,     17,
        16,     15,     14,     13,     12,     11,     10,     9,
        8,      7,      6,      5,      4,      3,      2,      1,
};

int exchange(int i)
{
    return tbl[i];
}
```

この変換に使う配列を「テーブル」と呼び，それを利用するコーディング方法を「テーブル参照」といいます。「テーブルから値を引く」ともいいます。「引く」とは「引っ張ってくる」の意味合いがあります。利点として「高速化」「コード量の圧縮」「繁雑なコードの整理」などがあります。複雑な計算式を使っていて処理速度が遅くなってしまったようなときに，高速化のために使われるテクニックです。

この例で示したのは，ごく簡単な処理なのでテーブルを使う利点はありませんが，テーブルの応用範囲はとても広く，あちらこちらで使われています。圧縮方法の1つであるLZW方式などは，辞書という名前でテーブルを圧縮技法の中心として利用しています。高速化するときは圧縮された符号データが取り得る範囲全部をテーブルに持たせたり，グラフィックならピクセルの色変換やピクセルを移動させるときの座標変換などに使われます。

◆値以外のものも格納できる

テーブルに格納する「変換後の値」には単純な値以外にもさまざまなものを入れることができます。たとえば値のかわりにポインタを格納すると，それに対応するデータを得ることができます。文字列を指すポインタを格納しておけば，変数と対応する文字列の取得が簡単にできます。

このポインタで構造体を指すと，複数存在するデータをテーブルで管理することができます。こうすると，1つの変数から複数のデータを引き出すことができます(List 1-4)。これはマップのイベント管理やキャラクタのパラメータ管理などによく利用されています。複数のデータを1つにまとめて管理することは，とてもたいせつなことです。これからも何度か出てくるので，この使い方は覚えておいてください。

List 1-4●ポインタで構造体を指す(Delphi)

```
{ 0～3の範囲の値に対応する構造体を取得する関数 }

{ 型宣言 }
type
    rec = record
        n : integer;
        str : string[80];
    end;


{ 定数宣言 }
const
    tbl : array[0..3] of rec = (
        (n : 0; str : '0 番目の構造体'),
        (n : 1; str : '1 番目の構造体'),
        (n : 2; str : '2 番目の構造体'),
        (n : 3; str : '3 番目の構造体')
    );


{ 関数本体 }
{ 出力変数に var を付けることで構造体のアドレスを渡すことができる }
procedure ExchangeRec(i : integer; var outrec: rec);
begin
    outrec := tbl[i];
end;
```

◆テーブル参照の問題点

テーブル参照にはよく陥る問題があります。「域外参照」と「データ用メモリの不足」の2点です。

テーブルには変換する値の「範囲」が必ずあります。これを超えてテーブルを参照しようとすると「域外参照」が起こります。これが起こると，変換された値がでたらめなものになるばかりでなく，データ領域そのものが破壊されたり，その結果としてプログラムが動かなくなることもあります。この予防のために変換する値を事前にチェックして，テーブルが持つ値の範囲を超えないようにするとよいでしょう。

あまりに大きなテーブルを作ると「データ用メモリが不足」することがあります。「変換したい値」が0～65,536の範囲にあるとき，テーブルのデータはその範囲の最大の数「65,536」個まで用意しなくてはいけません。もし変換される結果が4バイトの大きさになるのなら「65,536 × 4 = 262,144バイト」もメモリが必要になります。これだけ大きいとデータ領域をかなり圧迫してしまいます。

そこで，テーブルデータの数を少なくする処理が必要になります。そのためには，「変換したい値」や「テーブル」を16進法のビット幅で何個かに分割して処理するようにします(Fig. 1-2)。こうするとテーブルに使うデータの個数を大幅に減らすことができます。

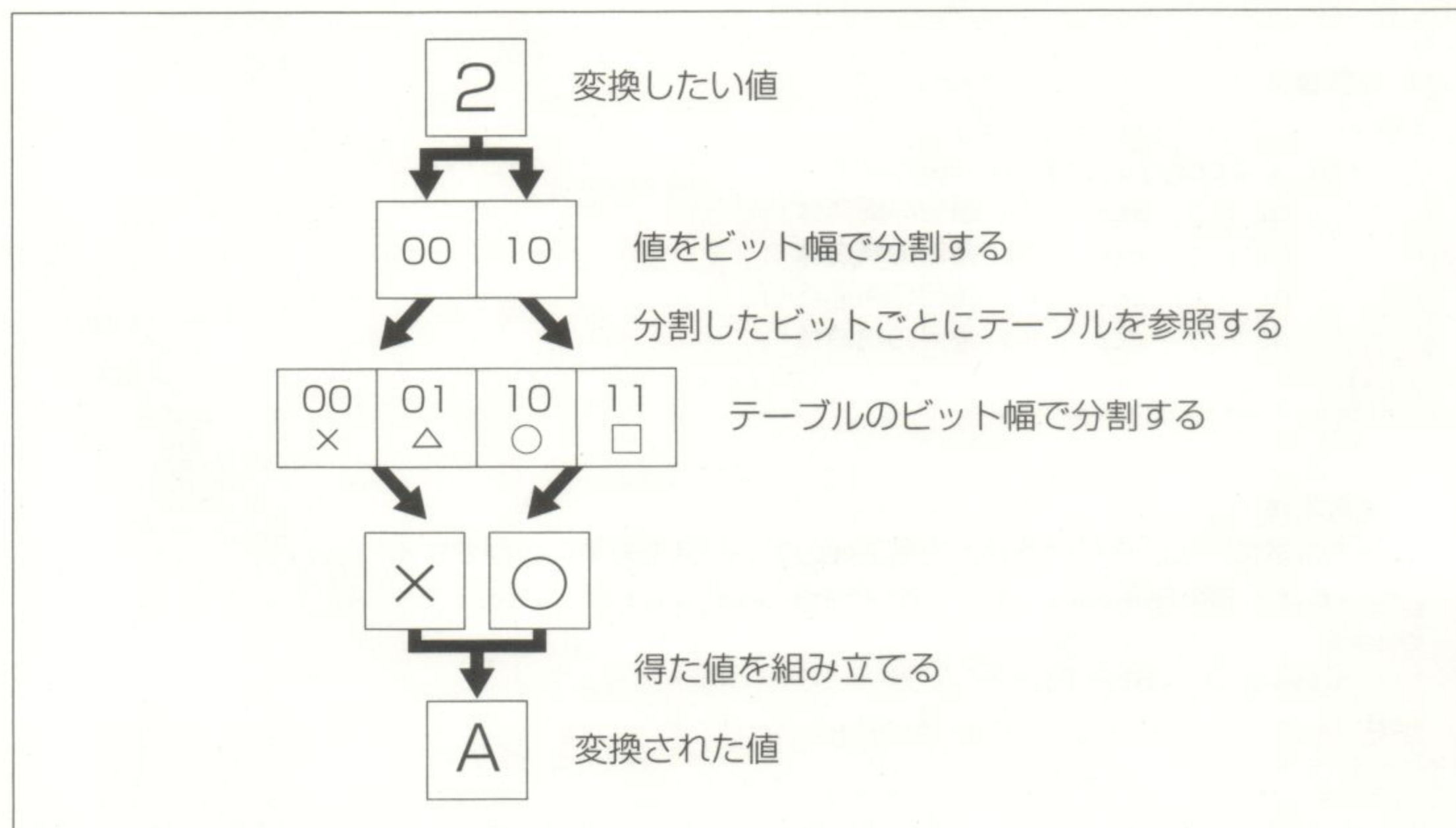



Fig. 1-2●変換したい値とテーブルを16進法のビット幅で分割することにより，用意するテーブル数を減らすことができる(この図では2ビット幅に分割)

たとえば，テーブル範囲の最大数が65,536個の場合，4ビット幅に分割することでテーブルに使うデータは4ビットの最大値となる16(0x1111)個だけで済むようになります。値を変換するときは「変換したい値」をこのビット幅に分割してから値を引き出します。得られた値は分割したときの順序に従って組み立てます。

テーブルが小さくても，List 1-2のように静的なデータとしてテーブルをたくさん作ると，場合によっては「データヒープが足りない」というコンパイラエラーが出ることがあります。これは静的データが多くありすぎるのが原因です。こんなときはメモリを動的に確保して，そこにテーブルを作るようにします。

● 符号データ

飲食店や遊楽施設では，従業員同士の暗号があるそうです。大きな声で「ありがとうございました」というと，それがほかの従業員へ助けを求める意味で使われたり，ゴキブリやお手洗などの言葉は，お客へ不快感を与えない別の言葉に置き換えられています。「符号」もこれと同じです。符号というと意味が広いのですが，たとえば「圧縮符号」では，「圧縮した結果から作られたデータ」のことを指します。もし「aなら32バイト，bなら16バイト，……，nならxバイト前の位置にあるデータを決まった数だけコピーする」といった圧縮方法の場合，符号とは「a，b，……，n」の部分に当たります。

先ほどのテーブルでも，この「符号」という言葉が使われることがあります。「aをテーブルから参照すると1が得られる」というときに，このaは符号ということもできます。このように符号とは，何かを示すデータの1つということになります。

● 関数ポインタ

日本のプロサッカーリーグであるJリーグでは，野球と同じように選手と背番号が一致する，固定背番号制が取られました。でも，一部の国のプロリーグを除いて，基本的にはサッカーの背番号は選手名と一致しません。サッカーの背番号はどちらかというとポジションを表しています。たとえば，あるチームのフォワードの背番号は「11」ですが，試合によっては，その背番号を付けるのは「岡野」だったり「福田」だったりします。

プログラムでも「背番号」は共通にして，それを使う「選手」を入れ換えることができたら，何かと便利です。たとえば「foo」という関数はどこからでも共通

に呼び出せますが，呼び出し先によっては「bar」という動きをして，また別のところでは「hoge」という動きをさせることを考えてみましょう。

こうした動きをする関数を普通に作ると，foo関数の中で，ある変数の値に従ってbarやhogeの処理を呼び出すような形になると思います。この変数を操作すれば，前述のような動きを作り出すことができます。でも呼び出す関数や判定する変数の数が多くなると，条件分岐文(ifやcase)だらけになってしまいます。これではバグが入る下地を作っているようなものです。

条件分岐を減らすには，前述のテーブルを活用します。関数を並べたテーブルを用意することができるのなら，変数からすぐに対応する関数を呼び出すことができます。データではポインタの形にしてテーブルに格納する方法がありました。ポインタというのは「何かを指し示している」もののことです。普通は文字列や構造体で利用していますが，データが指し示せるのなら同じように「ある関数を指し示すポインタ」も作ることができるはずです。

◆関数にアドレスを付ける

プログラミングの世界では，関数は関数名を使って呼び出しています。fooという関数を呼び出すには，fooという文字列を使わなくてはいけません。でも実際は，コンピュータもサッカーと同じように，背番号で関数を呼び出しているのです。Fig. 1-3は実行形式で使われるメモリモデルの例です。実行形式は，通常「データ」，「コード」，「スタック」の3つの部分に分かれています。「コード」にはCPUが直接解釈して実行する命令(ニモニック)が入っています。コンパイラが関数をコンパイルすると，このコードエリアへニモニックに変換した関数が格納されます。同時に関数を格納した「順番」とニモニックの「量」を基にしてその関数の背番号を作ります。「始めの関数が1番，その次は……」というようにです。こうして「foo関数は背番号○○」と背番号が決められます。ほかの関数からfooが呼び出されたときは，コンパイラが関数名をこの背番号へ置き換えて処理します。CPUはこの背番号から各関数を見つけ出して，関数の命令を実行しています。

この背番号が「アドレス」です。このアドレスはコードとまったく同じようにデータにも存在します。アドレスがわかればポインタを作ることができます。なぜならポインタが格納している値は，このアドレスそのものだからです。ポインタによってデータを取り出すときは，ポインタが指し示しているアドレスからデータを持ってきます。これが関数ポインタだったら，CPUは「ポインタに示されたアドレスにジャンプ」し，そのアドレスの関数が実行されることになります(Fig. 1-4)。

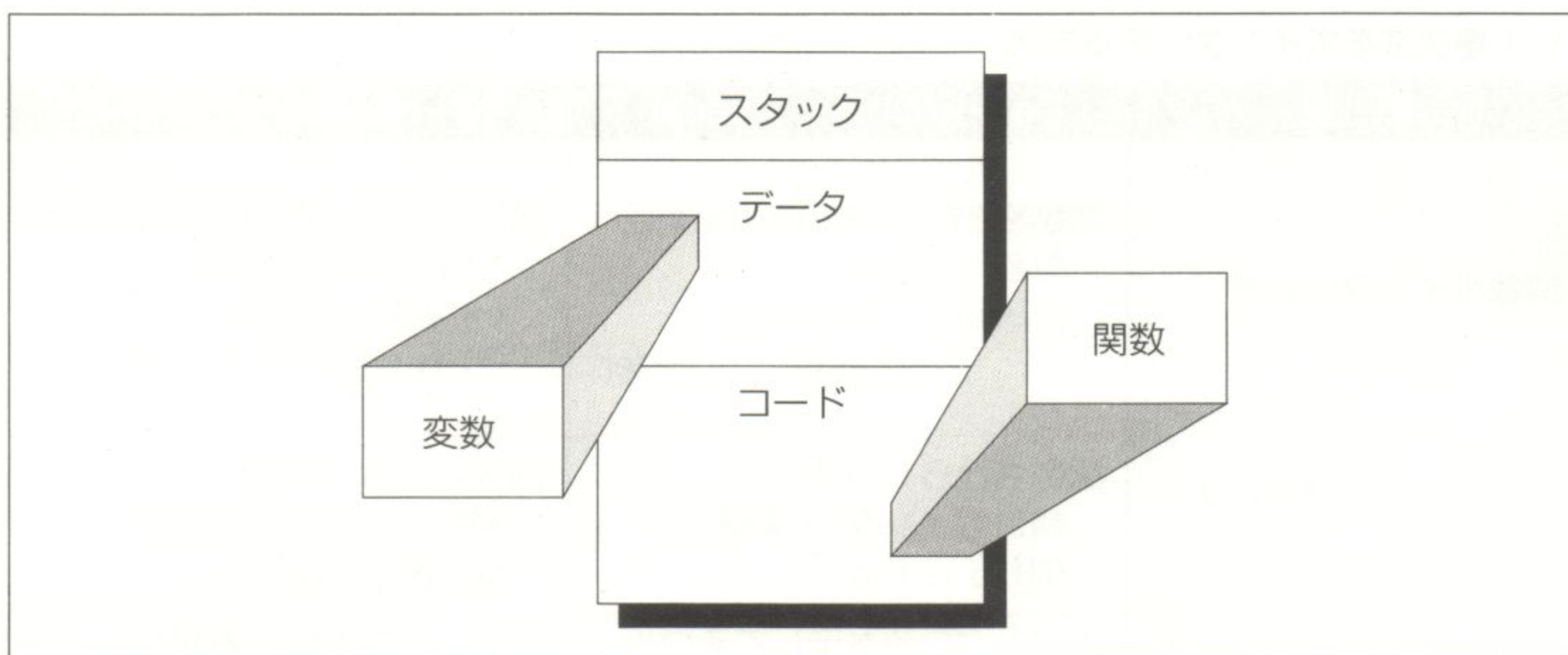



Fig. 1-3 ●実行形式で使われるメモリモデルの例。コード部分に関数が収録され，データ部分に変数が収録される

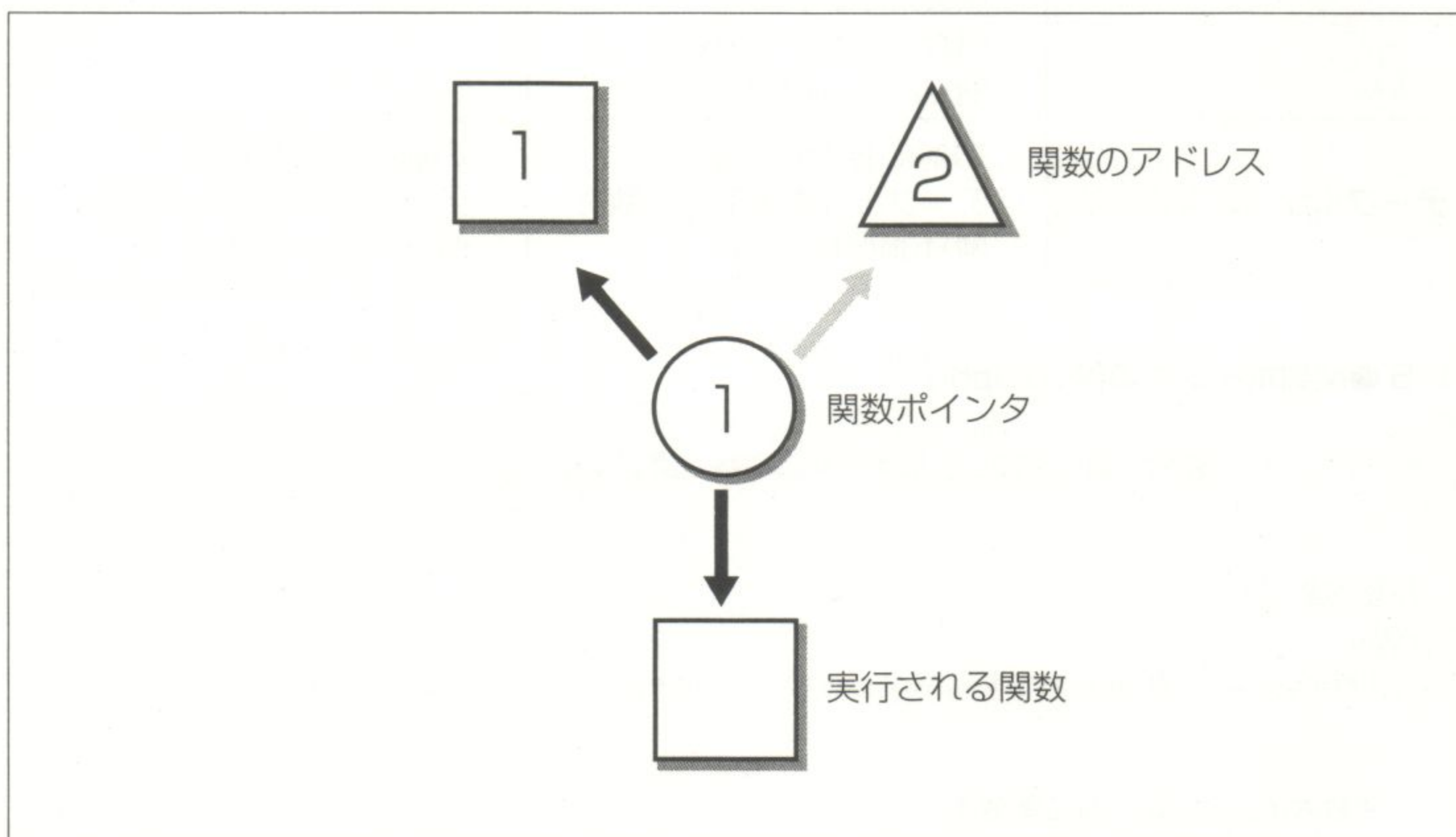



Fig. 1-4 ●関数ポインタのモデル。関数ポインタに示されたアドレスの関数が実行される

◆イベントハンドラを利用する

関数をポインタにする方法は言語によって違います。それぞれ代表的な方法をTable 1-1にまとめました。基本的な方法は，「関数を示す型」を用意して，「その型のポインタ」を作ります。あとはそのポインタに「関数のアドレス」を代入してやれば，関数ポインタのできあがりです。List 1-5，1-6は先ほどのfooのように，実行する関数を引数ごとに選択する関数の例です。

Table 1-1 ●関数をポインタにする方法

機　能	Delphi（Object Pascal）	C++Builder（C/C++）
関数ポインタ型宣言	関数名を外したものを指定する 型名 = 関数型; 例）func = function (i: integer) : integer;	関数名に当たる部分が型名になる typedef 返り値の型 （＊型名）（引数の型）; 例）typedef int（＊func）（int）;
関数ポインタアドレス代入	関数名から () を取る 関数ポインタ := 関数名; 例）p := foo;	関数名から () を取る 関数ポインタ = 関数名; 例）p = foo;
関数ポインタ実行方法	ポインタの値を示してそのまま書く 引数が必要なときは () を付ける 関数ポインタ; 関数ポインタ（引数）; 例）p^;　p^（0）;	ポインタの値に () を付ける ＊関数ポインタ (); 例）＊p ();
テーブルからの実行方法	() の前に [] を付ける テーブル [添え字]（引数）; 例）functbl [i]（i）;	() の前に [] を付ける テーブル [添え字]（引数）; 例）functbl [i]（i）;

List 1-5 ●関数ポインタの例（Delphi）

```
{  0 ～ 3 の範囲の値で引数ごとに実行する関数を選択する  }


{ 型宣言 }
type
   Funcptr = function(i : integer) : integer;


{ 関数ポインタで示される関数達                    }
function ExecZero(i : integer) : integer;
begin
   ExecZero := 0;
end;

function ExecOne(i : integer) : integer;
begin
   ExecOne := 1;
end;

function ExecTwo(i : integer) : integer;
begin
   ExecTwo := 2;
```

```
end;

function ExecThree(i : integer) : integer;
begin
   ExecThree := 3;
end;


{ 関数ポインタのテーブル                                   }
const
   Functbl : array[0..3] of Funcptr = (
      ExecZero,
      ExecOne,
      ExecTwo,
      ExecThree
   );


{ 関数ポインタから関数を実行                               }
{    引数:      実行する関数                               }
{    返り値:    実行された関数の返り値                     }
function ExecProcpointer(i : integer) : integer;
begin
   ExecProcpointer := Functbl[i](i);
end;
```

List 1-6 ●関数ポインタの例（C/C++）

```
/*  関数ポインタを使った関数           */

/* 型宣言 */
typedef int (*funcptr)(int);

/* 関数ポインタで示される関数達          */
int exec_zero(int i)
{
   return 0;
}

int exec_one(int i)
{
   return 1;
}

int exec_two(int i)
```

List 1-6

```
{
    return 2;
}

int exec_three(int i)
{
    return 3;
}


/* 関数ポインタのテーブル                          */
static funcptr functbl[] = {
    exec_zero,
    exec_one,
    exec_two,
    exec_three,
};


/* 関数ポインタから関数を実行                      */
/*    引数:      実行する関数                      */
/*    返り値:  実行された関数の返り値              */
int exec_procpointer(int i)
{
    return functbl[i](i);
}
```

オブジェクト指向の開発環境であるC++BuilderやDelphiでは，この仕組みを提供しています。よくOnClick, OnCreateなどの名称で使っている「イベント」というものがあります。これらは「関数をプロパティへ指定することで，特定のイベントが起きたときにその関数が呼び出される」という処理を行っています。この「イベントハンドラ」と呼ばれる仕組みが，関数ポインタとして使えるのです。

まず使いたい関数の型を用意して，この関数のあとに，Delphiなら「of Object」を置きます。C++Builderなら「__closure *」を関数名の直前に付けて宣言します。あとはクラスの宣言部分にこの型を使ったプロパティの形でポインタを用意するだけです(Table 1-1参照)。

言語は違っても，そう大きく方法は変わりません。配列にしたいときは，プロパティから呼ばれる下請け関数を先に作って，このなかで関数ポインタをまとめたテーブルから実際の関数を呼び出してやります。テーブルのほうはそれを初期化する専用の関数を作っておきます。List 1-5，6をこの方法で書き直したのがList 1-7, 1-8です。もちろんテーブルから直接参照するようにしても利用できます。

List 1-7●イベントハンドラで関数ポインタを実現する(Delphi)

```
{  関数ポインタを使った関数  }


unit Unit1;

interface

uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls,
                                             Forms, Dialogs;

{ 型宣言 }
type
   Funcptr = function(i : integer) : integer of object;

   TForm1 = class(TForm)
      procedure FormCreate(Sender: TObject);
   private
      Functbl : array[0..3] of Funcptr;
      function  ExecZero(i : integer) : integer;
      function  ExecOne(i : integer) : integer;
      function  ExecTwo(i : integer) : integer;
      function  ExecThree(i : integer) : integer;
      procedure exec_init;
      function  procpointer(ofs : integer; i : integer) : integer;
      property exec_procpointer[ofs: integer; i: integer]:
                                     integer  read procpointer;
   public
      { Public 宣言 }
   end;

var
   Form1: TForm1;

implementation

{$R *.DFM}


{ 関数ポインタで示される関数達                              }
function TForm1.ExecZero(i : integer) : integer;
begin
   Result := 0;
end;

function TForm1.ExecOne(i : integer) : integer;
begin
   Result := 1;
```

List 1-7

```
end;

function TForm1.ExecTwo(i : integer) : integer;
begin
   Result := 2;
end;

function TForm1.ExecThree(i : integer) : integer;
begin
   Result := 3;
end;

{ 関数ポインタリスト初期化用 }
procedure TForm1.exec_init;
begin
   functbl[0] := ExecZero;
   functbl[1] := ExecOne;
   functbl[2] := ExecTwo;
   functbl[3] := ExecThree;
end;

function TForm1.procpointer(ofs : integer; i : integer) : integer;
begin
   Result := functbl[ofs](i);
end;


procedure TForm1.FormCreate(Sender: TObject);
begin
   exec_init;
   Caption := IntToStr(exec_procpointer[3, 0]);
end;

end.
```

List 1-8 ●イベントハンドラで関数ポインタを実現する（C/C++）

```
/*  関数ポインタを使った関数  */


/* ヘッダファイル */
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
```

```
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
/* 型宣言 */
typedef int __fastcall (__closure *funcptr)(int);

class TForm1 : public TForm
{
__published:    // IDE 管理のコンポーネント
    void __fastcall FormCreate(TObject *Sender);
private:    // ユーザ宣言
    funcptr functbl[5];
    int __fastcall exec_zero(int i);
    int __fastcall exec_one(int i);
    int __fastcall exec_two(int i);
    int __fastcall exec_three(int i);
     int __fastcall procpointer(int ofs, int i);
    void __fastcall exec_init(void);
     __property funcptr exec_procpointer[int][int]
                                  = {read=procpointer};
public:             // ユーザ宣言
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------------
#endif


/* ユニットファイル */
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
/* 関数ポインタで示される関数達 */
int __fastcall TForm1::exec_zero(int i)
{
    return 0;
}
```

List 1-8

```
int __fastcall TForm1::exec_one(int i)
{
    return 1;
}

int __fastcall TForm1::exec_two(int i)
{
    return 2;
}

int __fastcall TForm1::exec_three(int i)
{
    return 3;
}

//---------------------------------------------------------------
/* 関数ポインタリスト初期化用 */
void __fastcall TForm1::exec_init(void)
{
    functbl[0] = exec_zero;
    functbl[1] = exec_one;
    functbl[2] = exec_two;
    functbl[3] = exec_three;
}

int __fastcall TForm1::procpointer(int ofs, int i)
{
    return functbl[ofs](i);
}

//---------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    exec_init();
     Caption = IntToStr(exec_procpointer[3][0]);
}
//---------------------------------------------------------------
```

◆関数ポインタの利点

関数ポインタには，「データと関数の対応ができる」「処理の切り換えが簡単になる」といった利点があります。とくに関数を何かのデータに対応させたいときに，よく使われます。単独で使うよりはテーブルや構造体などに含めて使うほうが便利です。注意しなくてはいけないのは「アドレスの値を間違えない」ことです。間違うとプログラムは確実にフリーズします。これだけは気をつけてください。

「別のプログラムから自分のプログラムが持っている関数を使いたい」というときには，関数ポインタにして渡せば目的の関数を実行できます。関数ポインタとテーブルを組み合わせると「このシーンではあるひとつの処理を実行する」といったシーンごとに対応する処理が簡単に実現できます。たとえば「functbl[scene](i);」というように現在のシーンを示すscene変数をテーブルに与えるだけで，対応する処理を呼び出せます。また，「次のシーン」へ移動するときはscene変数の値を変更するだけで済んでしまいます。if文などの条件分岐を使った方法と比べると，非常に楽に管理することができます。なお，こうした処理はシーンという状態が変わるところから「状態遷移」などと呼ばれます。

連鎖するデータ

「ある値を参照したとき，それに対応する別の値を知りたい」というように，データを連鎖させたいことがあります。Fig. 1-5のように，アドベンチャーゲームによく見られるツリー状の分岐や迷路のようなマップでは，普通にプログラムを作ると条件分岐の山になってしまいます。

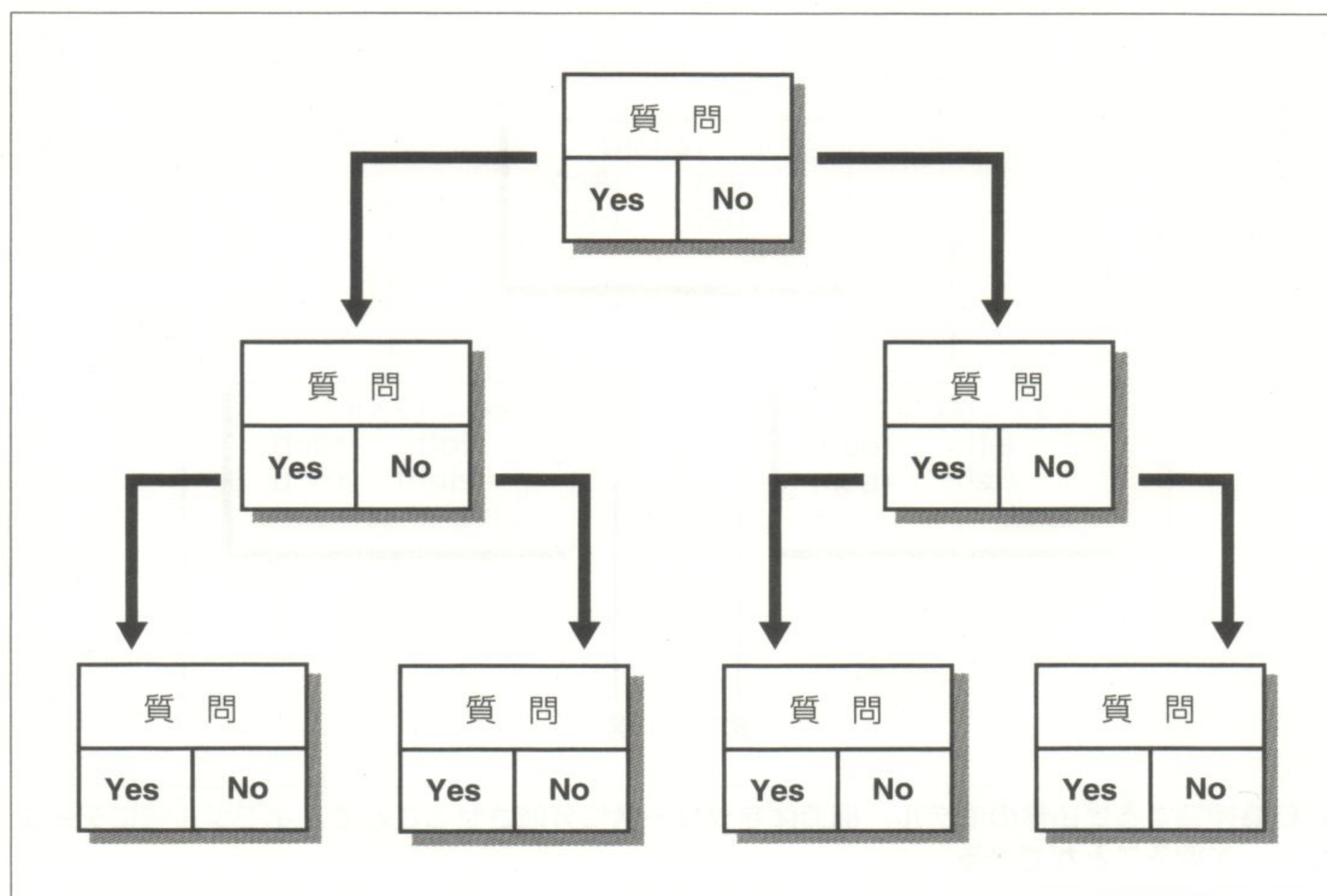



Fig. 1-5●ツリー状のデータ分岐のモデル。1つの質問に回答することで次の質問に進むという具合にデータが連鎖していく

◆構造体を用意する

こんなときは，始めに1つの場面で基本となる「構造体(セル)」を用意します。たとえばアドベンチャーゲームなら，「その場所を説明するメッセージ」，「その場所のグラフィックデータ」といったものです。こうしたものを全部まとめて構造体に定義します。これができたら「構造体自身を指し示すポインタ」を構造体に分岐する数だけ定義します。各言語でポインタの宣言方法が異なりますが，Table 1-2に代表的なものをまとめました。あとはそれぞれの場面の構造体をこのポインタに順序どおりに結合してやればよいのです。

Table 1-2●連鎖するデータの作り方

Delphi (Object Pascal)	C++Builder (C/C++)

初めにポインタ型を宣言する構造体にはこのポインタ型を使ってポインタを宣言する
type
　ポインタ型名 : ^構造体名;
　構造体名 = record
　　ポインタ名 : ポインタ型名;
　　データ;
　end;
	「struct 構造体名」を宣言したあとに「struct 構造体名 ＊ポインタ名」を構造体に含めることができる
	　typedef struct 構造体名 {
	　　struct 構造体名 ＊ポインタ名;
	　　データ;
	　};

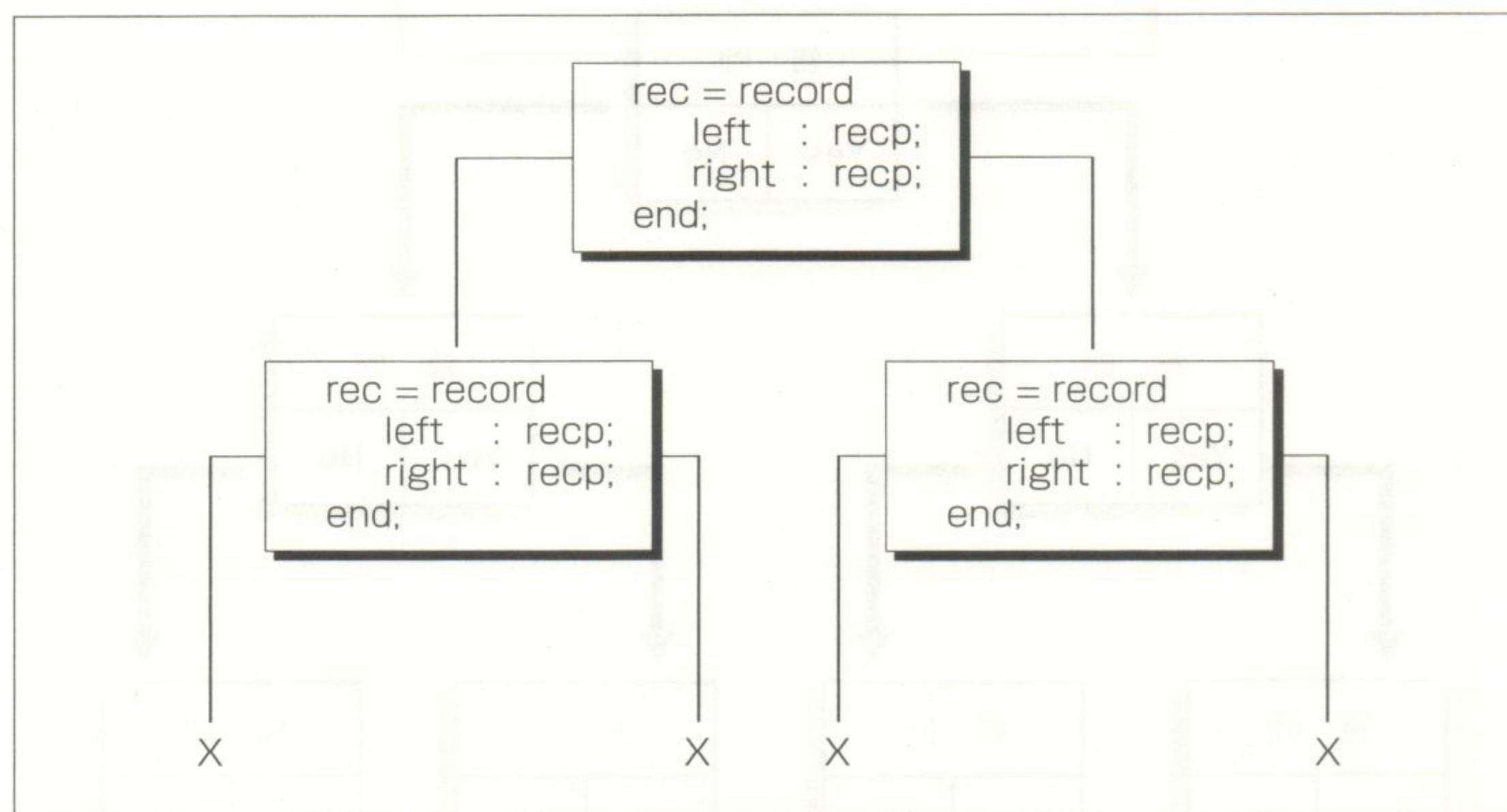



Fig. 1-6●連鎖する構造体のモデル。構造体をツリー状に分岐させることで，イモヅル式にデータを得ることができる

リストの終端に当たる構造体には，識別を簡単にするため，ポインタにNIL(NULL)を入れておきます。こうして作った構造体のリストは，先頭からポインタをたどっていくことでイモヅル式にデータを得ることができるのです(Fig. 1-6)。

データを検索する際には，1つのポインタを使ってアクセスします。始めは先頭の構造体を指すように設定します。始めの構造体の処理が終わったら，次の構造体を指す値をこのポインタに設定します。あとはこれを終端まで繰り返します(Fig. 1-7)。ポインタに対して処理を行うことで，終端のチェックや構造体へのアクセスといったことを1つにまとめられます(List 1-9，1-10)。

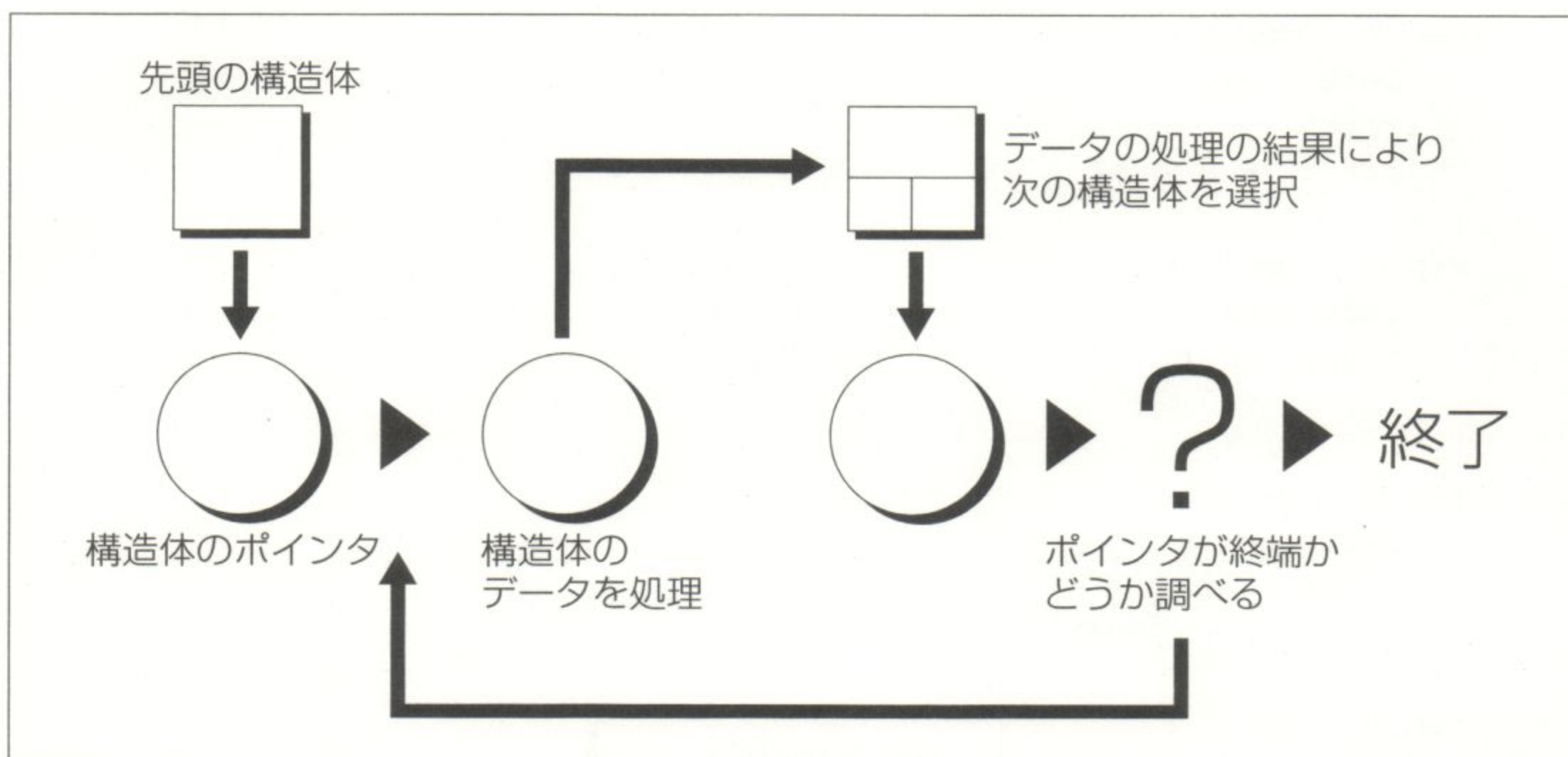



Fig. 1-7●構造体のアクセス方法。先頭の構造体の指すように設定したポインタで構造体を処理し，その結果により次の構造体を選択してポインタに設定する

List 1-9●連鎖した構造体(Delphi)

```
{  連鎖した構造体の例                                      }
{  2つずつ3段階に分岐する                                  }
{                                                          }
{                       recA                               }
{                    |        |                            }
{                 recB       recC                          }
{                |     |     |     |                       }
{              recD recE recF recG                         }


{ 型宣言 }
type
   recp = ^rec;
   rec = record
      left : recp;
```

List 1-9

```
      right : recp;
      str : string[80];
   end;

{ 定数宣言 }
{ 各構造体の宣言 }
const
   rec_g : rec = (
      left : nil;
      right : nil;
      str : 'G の構造体'
   );
   rec_f : rec = (
      left : nil;
      right : nil;
      str : 'F の構造体'
   );
   rec_e : rec = (
      left : nil;
      right : nil;
      str : 'E の構造体'
   );
   rec_d : rec = (
      left : nil;
      right : nil;
      str : 'D の構造体'
   );
   rec_c : rec = (
      left : @rec_f;
      right : @rec_g;
      str : 'C の構造体'
   );
   rec_b : rec = (
      left : @rec_d;
      right : @rec_e;
      str : 'B の構造体'
   );
   rec_a : rec = (
      left : @rec_b;
      right : @rec_c;
      str : 'A 番目の構造体'
   );


{ 関数本体 }
procedure ChainRecord;
var
   main_recp : recp;    { メインポインタの宣言  }
begin
```

```
    main_recp := @rec_a;
            { メインポインタに最初の構造体を設定 }
    while main_recp <> nil do begin
            { メインポインタが nil なら終了 }

        { 構造体のデータを処理 }

        { どちらの方向に移るかを入力 }
        if nextsw = LEFT then begin

            { 左に移るのなら }
            main_recp := main_recp^.left;

        end else begin

            { 右に移るのなら }
            main_recp := main_recp^.right;

        end;
    end;
end;
```

List 1-10●連鎖した構造体(C/C++)

```
/*　連鎖した構造体の例　*/


/*　型宣言　*/
typedef struct rec_t {
    struct rec_t *left;
    struct rec_t *right;
    char *str;
};


/*　各構造体の宣言　*/
struct rec_t rec_g = {
    NULL,
    NULL,
    "G の構造体",
};
struct rec_t rec_f = {
        NULL,
        NULL,
        "F の構造体",
};
struct rec_t rec_e = {
    NULL,
```

List 1-10

```
    NULL,
    "E の構造体",
};
struct rec_t rec_d = {
    NULL,
    NULL,
    "D の構造体",
};
struct rec_t rec_c = {
    &rec_f,
    &rec_g,
    "C の構造体",
};
struct rec_t rec_b = {
    &rec_d,
    &rec_e,
    "B の構造体",
};
struct rec_t rec_a = {
    &rec_b,
    &rec_c,
    "A 番目の構造体",
};


/*  関数本体  */
void ChainRecord(void)
{
    struct rec_t *main_recp;    /*  メインポインタの宣言    *

    main_recp = &rec_a;    /*  メインポインタに最初の構造体を設定  */
    while (main_recp != NULL) {    /*  メインポインタが NULL なら終了  */

        /*  構造体のデータを処理  */

        /*  どちらの方向に移るかを入力  */
        if (next_sw() == LEFT) {

            /*  左に移るのなら  */
            main_recp = main_recp->left;

        } else {

            /*  右に移るのなら  */
            main_recp = main_recp->right;

        }
    }
}
```

◆連鎖構造を応用する

このデータ構造は，応用次第でさまざまな場面に活用できる方法です。たとえば構造体に前述の関数ポインタを含めれば，分岐先のイベント処理を追加することができます。また連鎖を切ったり外したりするのも，普通のデータ構造と比べたらとても簡単です。

構造体をファイルに格納するには，ポインタのかわりに「先頭から○番目の構造体」というように番号に直してから格納します。ファイルを使うときは，この番号からポインタへ変換する作業が必要です。

C++Builder/DelphiのTTreeItems, TTreeItemはこの考えを使ったクラスです。TTreeItemをセル(ノード)として扱い，自由に分岐させることができます。うまく使えばこうした構造体や処理をみずから用意しなくて済むでしょう。また，連鎖状態を表示するTTreeViewコンポーネントが用意されています。

◆無限ループ

ゲームというのは「繰り返す」ことばっかりしています。カードゲームではプレイヤーごとに順番が回ってくるでしょうし，シューティングゲームでは常に画面を書き直してキャラクタを動かすことが必要になります。ほとんどのゲームでは，ゲームが終わるまでこの繰り返しが続きます。繰り返しを実現するのが「無限ループ」です。C/C++なら「for(;;) { コード }」や「while(1){ コード }」，ObjectPascalなら「while True do begin コード end;」という形でずっと「コード」部分の処理を繰り返します。ゲームが終了してループを抜けたいときは，両言語とも「break;」を呼び出します。Windowsプログラムでは，Windowsから提供されるサービスの利用やほかのアプリケーションへの配慮から，無限ループ中にはメッセージ処理を行うAPIを呼び出すか，スレッドを作ってプロセスを分離するようにします。これらについては付録CD-ROMのソースコードを参照してください。

●グラフィックデータの転送

ゲームを作るうえで，もっとも巨大になってしまうのが，グラフィックデータです。800×600の画面サイズで256色の場合，480,000バイトにもなります。約0.5Mバイトです。これだけの大きさになると，VRAMへの転送にとても時間がかかります。高速性が求められるゲームでは，何らかの対処方法を考えないといけません。

プログラム側でできる対処法としては，「あらかじめメモリにグラフィックデー

タを置いておく」というのが基本になります。これがもっとも高速でちらつきがなく処理できるからです。よく使われるのが「画面バッファ」です。VRAMと同じサイズでメモリに確保されたバッファをこう呼びます。

次に「転送量そのものを小さくする」という方法があります。解像度や使う色数などを減らすと，結果的にデータサイズも減らせることになります。たとえば前述のデータなら，16色にすることで，半分の240,000バイトにすることができます。また画面を分割して，1回に転送するデータ量を減らすといった方法も使われます。

◆ピクセルの扱い

ピクセルとは「画面上の点」のことです。この点が集まって画面に絵を表示しています。パックドピクセルでは1ピクセルが連続したビットになっています。8ビットのデータで1ピクセルを表示できるのなら，色数は2^8で256色まで表示可能です。

Windowsのビットマップでは，パックドピクセルを使っています。ただしデータで最初のラインが画面に表示されるといちばん下のラインになる「上下逆さま」となっています。C++Builder/DelphiのTCanvasコンポネートにあるPixelsプロパティなど，描画に使うメソッド/プロパティでは，座標はそのまま逆さまにしないで使えます。ビットマップをダイレクトにメモリやファイルとして扱うときだけ，この状態になっていることを忘れないでください。

● パーツデータの管理

アクション系のゲームでは，背景とキャラクタとでデータの管理方法が違います。この場合の背景とは，シューティングゲームなどに使われる全画面に広がったドロー系の画面データを指します。こちらは普通のグラフィックデータと同じように管理することができます。一方キャラクタデータは，細かい部品(パーツ)に分けて格納しなければなりません。シューティングゲームの機体のデータなら機体だけ，敵キャラなら敵キャラだけ，というようにです。

パーツデータの管理はテーブルからデータを参照する方法が便利です。たとえば次の面で敵キャラを変えたいときは，テーブルに収めているアドレスを変えてやります。これならコード部分を変更しなくても，データだけを簡単に変えることができます。さらに敵キャラの動きなどもデータにしておいて，このパーツデータといっしょにまとめて管理してやると，関係するほとんどのコードをまとめられます。

◆パーツと背景の合成

パーツと背景の合成は，前述の画面バッファで行います。方法は「背景を透かせたい」部分を描いたデータ(マスクデータ)を別に用意し，これを背景と「AND」したあと，パーツデータを「OR」します。ANDはデータのビットが両方立っていれば(1ならば)フラグが立ち，ORはどちらかのビットが立っていればフラグが立ちます。データを数字の列にしてこの計算を見てみるとFig. 1-8のようになります。合成が終わった画面はVSYNCなどを使って，ある一定期間ごとに画面バッファからVRAMに転送していきます。Fig. 1-9は横にスクロールするゲームで使われている合成時のデータの動きです。

パーツが動くときには，いったんパーツを消して背景を元に戻す作業が必要です。消去方法にはいろいろありますが，メモリやCPUのパワーに余裕があるときは，合成する画面バッファとは別に背景を管理したほうが楽です。背景は合成をする前の段階で保存しておき，合成するときは全部の背景を画面バッファにコピーするのです。Fig. 1-9ではこの方法を使っています。そうでないときは，合成するときに背景をパーツの範囲だけメモリに待避して，次に合成するときにそれを戻すようにします。

```
背景データ          1111000110001111b (F18Fh)
キャラクタデータ    0000001001000000b (0240h)
マスクデータ        1111110000111111b (FC3Fh)

①背景 AND マスクデータ
           1111000110001111b
    and    1111110000111111b
    ----------------------------
           1111000000001111b (F00Fh)
                    ↓
②背景 OR キャラクタデータ
           1111000000001111b
    or     0000001001000000b
    ----------------------------
           1111001001001111b (F24Fh)
                    ↓
③合成完了
```

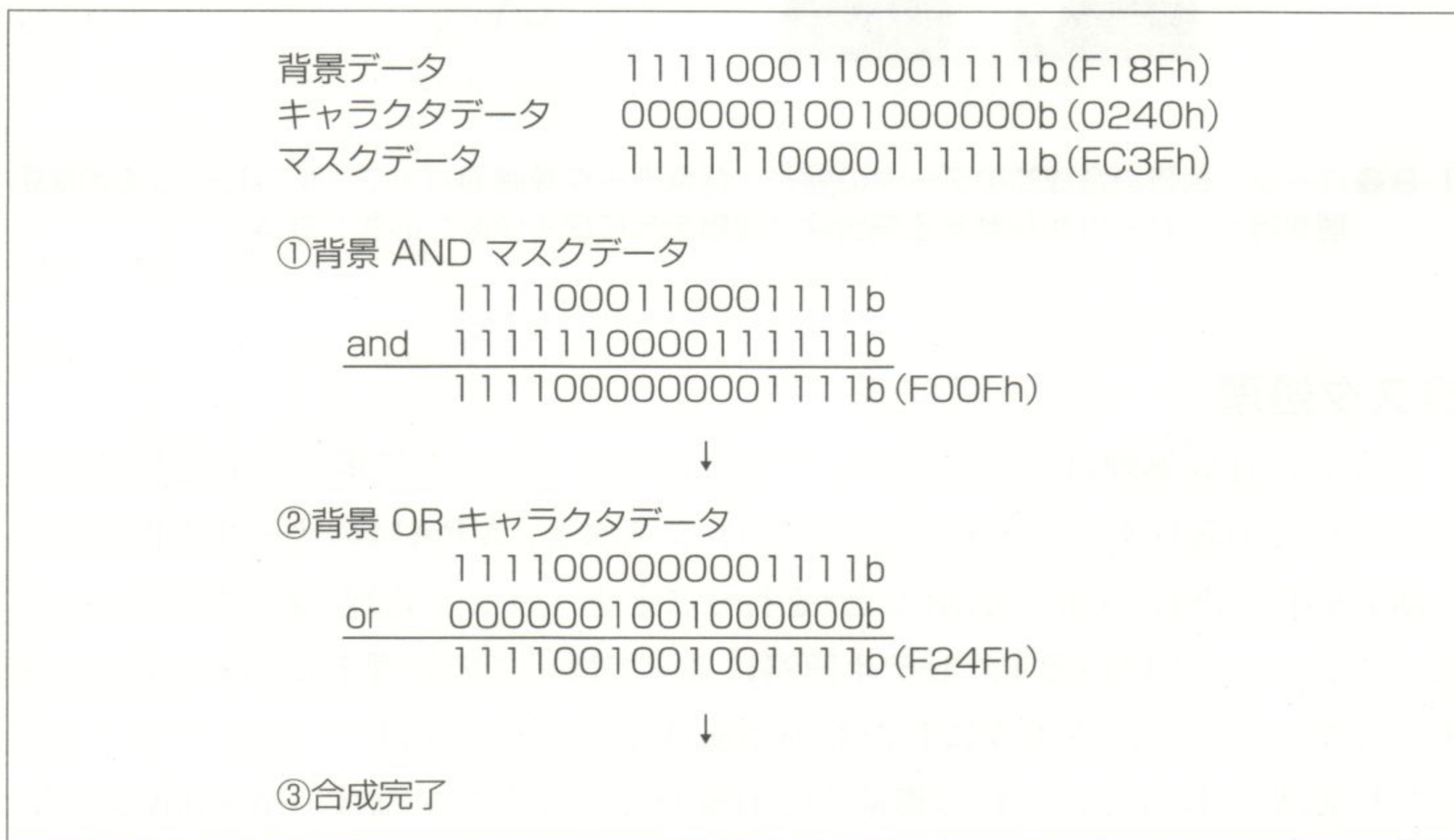



Fig. 1-8 ●データの合成の計算。背景データとマスクデータをANDした数値に，キャラクタデータをORすれば，背景とキャラクタが合成できる

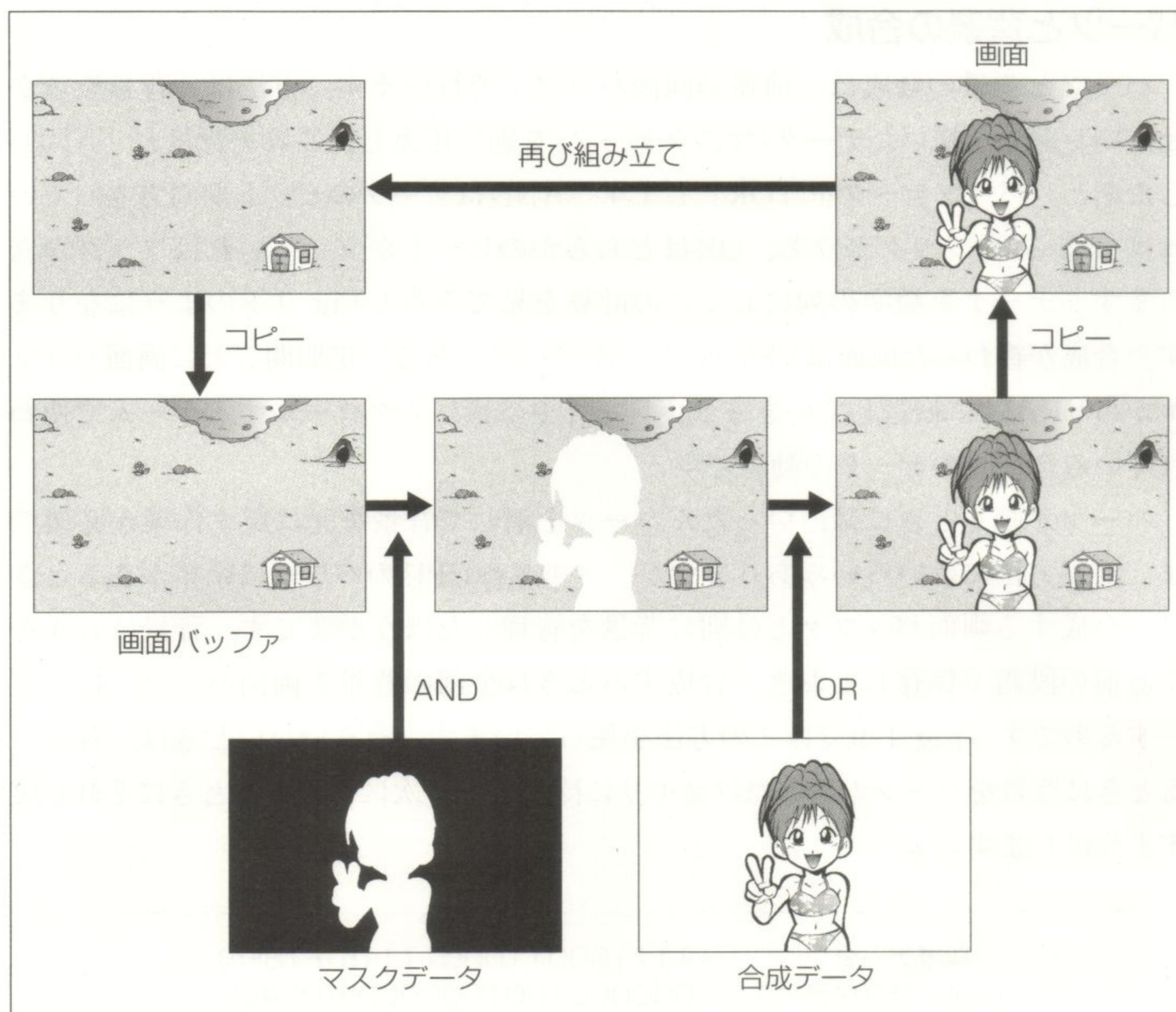



Fig. 1-9●パーツと背景の合成時のデータの動き。背景データを画面バッファにコピーして合成処理を行う。パーツが移動する場合は，背景を元に戻す作業が必要となる

◆ラスタ処理

こうした合成処理は，ハードウェアによっては「ラスタ処理」として用意されていることがあります。メモリやCPUのレジスタと，VRAMのデータの間でビット演算処理(AND，OR，XORなど)を行うことを「ラスタ処理(ラスタオペレーション：ROP)」といいます。ビデオ回路によっては，この処理をビデオ回路のハードウェアで処理できるようにしたものがあります。たとえばPC-9801シリーズでは，EGCやGRCGと呼ばれる機能がこれをサポートしています。Windowsでも，デバイスコンテキストから転送するときにこの処理が使えます。この場合はビデオドライバを介して，ヒデオボードの機能を呼び出すことになります。

C++Builder/Delphiでは，この合成処理にTImageListコンポーネントが利用できます。このコンポーネントを使うと，マスクデータを自動的に生成してくれた

り，背景との合成も簡単に行うことができます。これを利用した擬似スプライトの作り方は後述します。

ほとんどの場合でCPUによる演算よりも，高速に処理を行うことができます。ただし，うまく使わないとちらつきが出たりします。VRAMの裏画面を使えるようなものではそれを，そうでなければ画面バッファをメモリ上に確保して，そこで合成するようにして使うとよいでしょう。

● セーブ/ロード

かつて「ゲームブック」というものが流行ったことがありました。読み進んでいくに従って，途中で質問が出され，そのときの状態に合わせて指定の番号までページを飛ばして，物語を楽しむというものです。質問の答えによって物語の展開が変わるため，分岐する部分のページに指を挟んで元へ戻れるようにしておくのですが，話が進むと指が足りなくなってしまい，間違って本を落したりでもしたらもうたいへんです。わけがわからなくなります。

コンピュータのゲームではそういうことがありません。それはコンピュータ側で元のページに当たる「物語の位置」を記憶しておくことができるからです。記憶することを「セーブ」，記憶したものを呼び出すのを「ロード」といいます。

◆進行状況を１つにまとめる

この仕組みをプログラムで実現する方法を考えると少しやっかいです。「位置」はファイルに書き出せばとりあえずよいのですが，保存するデータは「位置」だけではありません。物語の主人公が持つアイテムなども保存しておかないと，次にロードされたときに「セーブされたときの状態」へ戻らなくなってしまいます。シューティングゲームでは自機の残り数，オプション，マップ上での位置，どのマップにいるのか，なども保存しなければなりません。

逆に考えてみると「セーブ/ロードができるようにゲームの進行状態を1つにまとめておく」必要があります。各進行状況を変数としてまとめて構造体のようにして管理し，それをセーブするようにします(Fig. 1-10)。ロードするときは，セーブされていた値が指定されただけで，ゲームが正しく進行するように各処理を作っておきます。このためには「かぎられた情報だけですべてが動く」ようにしなければなりません。たとえばキャラクタが画面にいる状態を保存したいときには，「キャラクタのマップ上での座標」と「キャラクタとマップの関係」の2つの数値

だけで事足ります。ロードするときは，まず2つの数値からマップのどの地点を画面に出力するかを計算します。次にキャラクタのパーツをセーブした座標に従って画面上に出力します。

このように画面出力用の各処理を座標を与えただけで目的のものを表示するようにしておきます。ほかの処理も同じように作っておくと便利です。

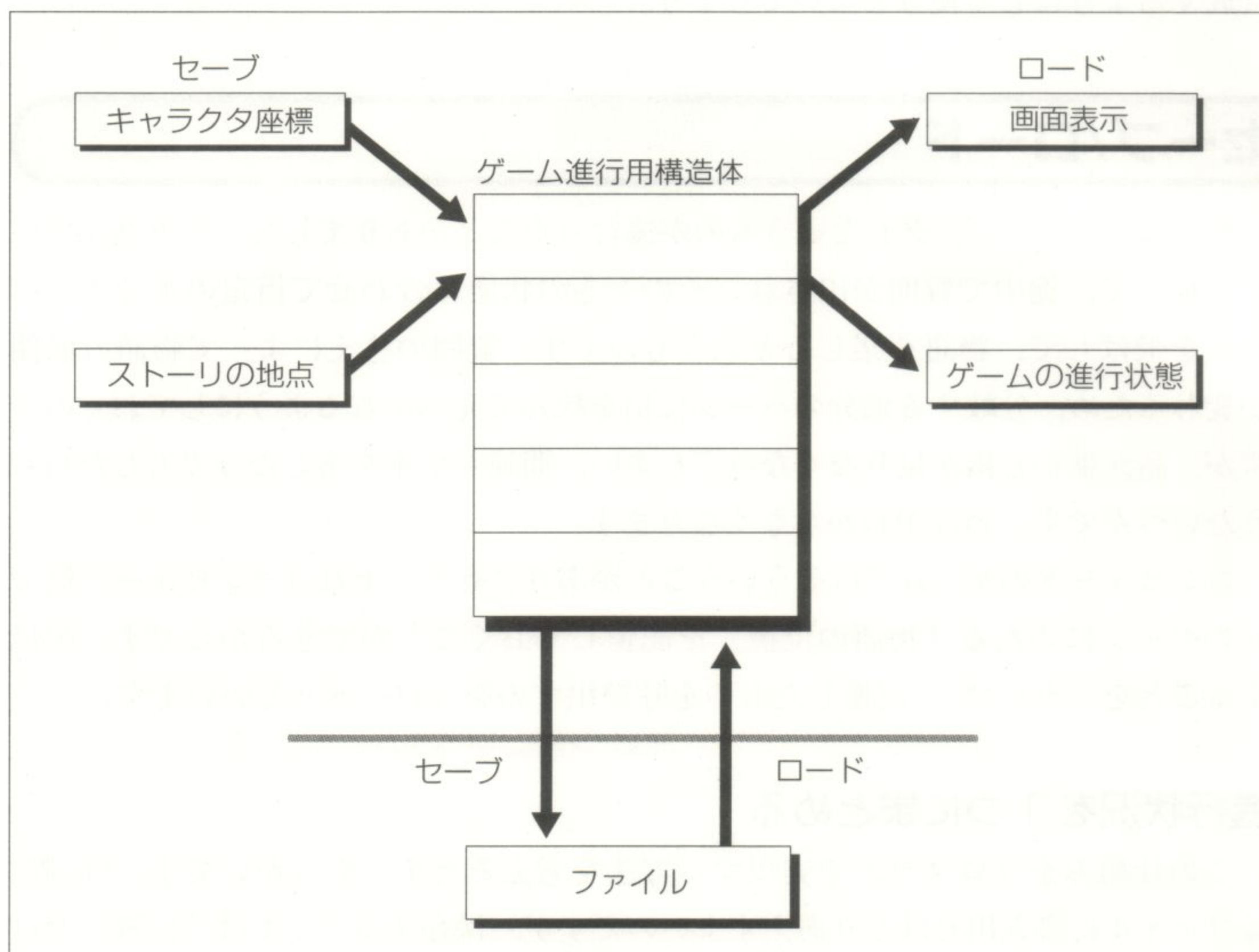



Fig. 1-10●セーブ時には，キャラクタの座標やゲームの進行状況を１つにまとめて構造体として管理してやる。ロード時には，その構造体のデータを基に画面表示やゲーム上に正しく進行状況を反映させる処理を行う

Section 3 さまざまな技術を利用する

ゲームのプラットホームとなるパソコンやコンシューママシンは，日進月歩の勢いで進化し続けています。そして，それに伴い新技術もつぎつぎと発表されています。さらに，Internetの普及による通信環境の整備のおかげで，マシンとマシンを回線でつないで対戦をするといった，ゲームの新しい楽しみ方も出てきました。ここでは，通信対戦とDirectXという新しい技術について解説をしてみたいと思います。これらは，ゲームをプログラムするうえで必ずしも必要というわけではありませんが，知っておいて損になることはないはずです。

● 通信対戦

ネットワーク環境の整備に伴って，「1人で楽しむゲーム」から「複数人で同時にゲームを行う」へと，ゲームの楽しみ方が変わってきました。それが「通信対戦」です。パソコンでは，すでにネットワーク環境がコンシューママシン(ゲーム専用機)よりも充実しているため，いくつか代表的なゲームが存在します。また，ハンディタイプのコンシューママシンでは，ケーブルを接続することでアイテムの交換ができるゲームもあります。

通信の目的は，そのゲームの性格やルールにより変わってきます。「格闘ゲーム」では「通信で結ばれた相手との対戦」が目的となりますが，「RPG」では互いのアイテムの交換など，いままでの「ただ相手を倒すだけ」から「相手と交渉する」タイプのものが増え始めています。まだ「ネットワークがなければゲームとして成り立たない」というところまで考えられたゲームは多くはありませんので，これからアイデア次第でもっとも発展する可能性がある分野です。

◆通信の方式

パソコンゲームでの通信対戦の環境としては「Internet/ネットワーク上でのゲーム」，「パソコン通信でのゲーム」，「ユーザ同士がモデム/シリアルケーブルなどで接続」などがあります。

もっとも古くからあったのは「シリアルクロスケーブルでの接続」です。2台のパソコンをRS-232Cなどのシリアルポートでクロスケーブル接続してゲームを行

います。これには2台のパソコンが必要で，しかもケーブルの長さに限度があるので，ごくかぎられた場所でしか遊べません。アーケードマシン(ゲームセンタなどにあるゲーム専用機)では，これと同じ方法が取られています。

最近では，Internet上のネットワークゲームが多くなりました。Javaなどネットワーク言語で組まれたゲームでは，ブラウザさえあればどの環境でも同じゲームができるなど，新しい面もあります。国や場所を選ばずにさまざまな人と遊べるというのは，とても楽しいものです。

ゲームを通信に対応させるためには，「通信そのものの技術」，「エラーやデータが届かなかったときの処理」，「ゲーム手順をどうデータとして送受信するか」など，これまでのゲーム作成とはあまり関係がなかったプログラミング分野の知識が必要です(Fig. 1-11)。

データによるゲームイベント起動
イベント発生時のデータ送信
データのパケット化
送信データの最適化
エラー送信の検出/再送
ゲーム本体
データの処理
通信処理
入力
表示
通信ドライバ
パケット
相手のゲーム
ドライバ操作
データ送受信
通信の接続

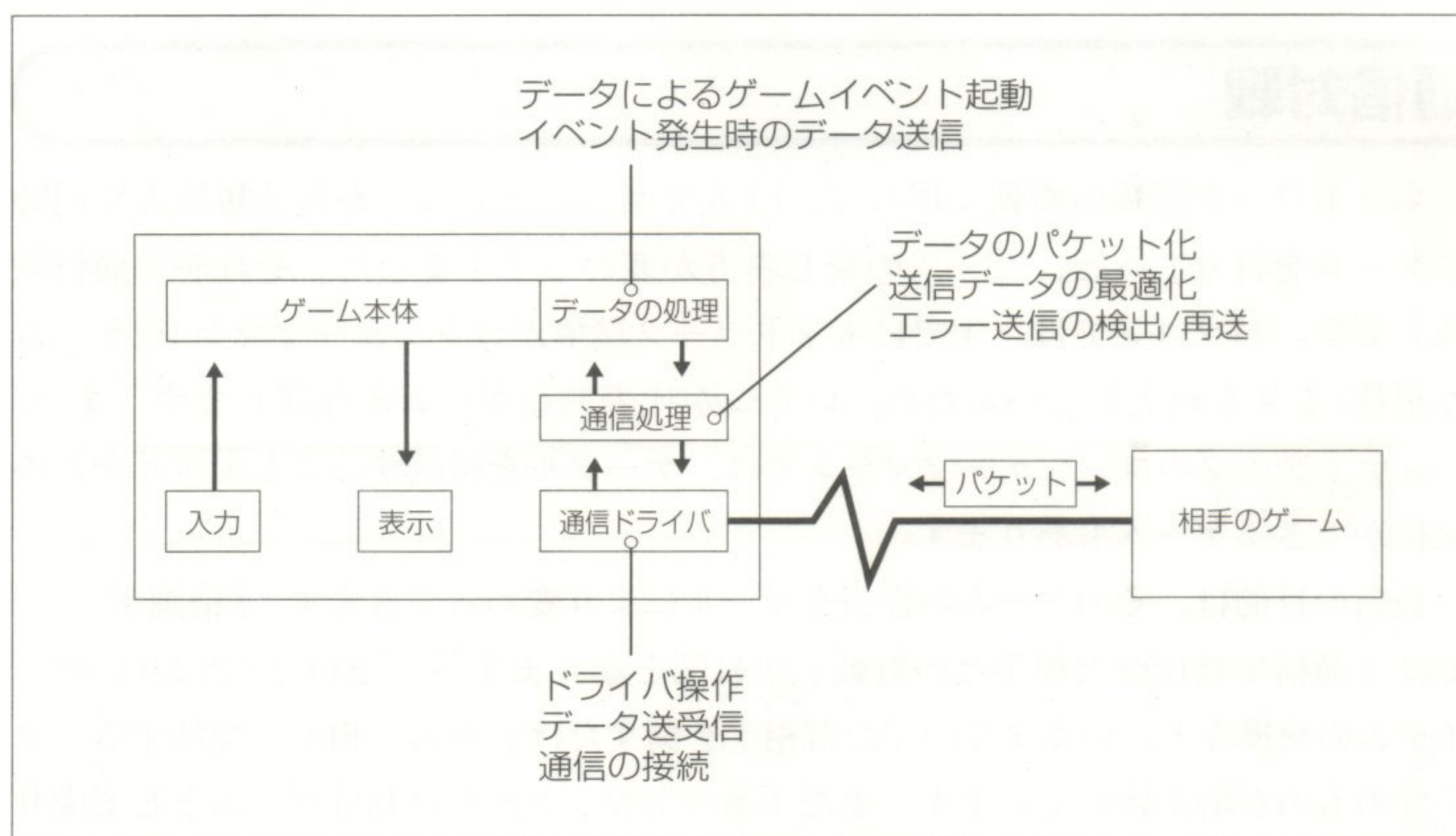



Fig. 1-11 ●データ処理/通信処理/通信ドライバなど，通信対戦への対応にはいままでにない技術が求められる

◆イベントをデータとして送る

送信されるデータは何らかのイベントを意味しています。プレイヤーが自機を操作して弾を撃ったのなら，「弾の種類」，「弾の座標」，「飛んでいく方向」などが対戦している相手にも必要になります。これらを1つのデータにして送り，受信側で「受け取った1つのデータから現在のゲームへ反映」します。たとえば「弾の発射」というデータがきたら，そのデータを基に相手側機の位置から座標や弾が動く方

向などをゲームに反映すればいいのです。このためゲーム処理に関しては「何らかのデータ/イベントが発生したら，それに合わせて画面表示やゲームを進める」という考えが基本になります。

これ以降の「弾をその方向へ進める」「弾が当たったかどうかの判定」などは，それぞれのゲームソフト側で処理します。即応性が必要なデータ/イベントは，このようにできるだけ通信せずに，受信側のソフトで処理するようにしています。

Internetでは，「送ったデータが必ず相手に届くとはかぎらない」ことが前提になっています。そのため，アイテムやゲーム進行の判定に必要なデータなど「確実に送れたかどうかの確認が必要なもの」，連射される弾など「確認が不要なもの」の2つを使い分けています。

通信の「速度」はたいへん遅いものです。最近はISDNなどデジタル通信が普及してきたとはいえ，ほかの機器と比べると圧倒的に遅いのです。また「速度が変動」します。Internetでは何らかの対策が施されていないかぎり，途中の経路が混雑しているとデータ転送が遅れます。スピードが求められるゲームでは，これらがネックとなることもあります。そこで「転送するデータをできるだけ少なくする最適化」など，この問題をどうクリアしていくのかが，ネットワーク対応ゲームを作成するためのポイントの1つです。

● DirectX

Microsoft社がWindows 95/98/MeやWindows NT/2000に用意した「ハードウェアを効率よく利用する」ためのドライバ群の総称を，「DirectX」といいます。このDirectXファミリは2001年5月現在，DirectX 8b(英語版)までリリースされています。これからもさまざまなデバイスに対応したり，より効率化を図るため，バージョンアップが続けられていく予定です。

最初のDirectXは「ゲーム用のAPI」という意味合いが強かったのですが，最近ではゲームだけではなく，マルチメディア全般の技術を提供しています(Table 1-3)。やがては「デバイスのコントロール」がすべて含まれてしまいそうです。アーケードマシンやコンシューママシンでの利用も将来的にはあるかもしれません。

◆DirectXの機能

では，DirectXに対応しているとゲームがどう変わるのでしょうか？　Direct3Dは，ビデオカードの3D機能をかぎりなくフルに引き出します。DirectDrawは，高

速表示やオーバレイ(合成，重ね合わせ)を提供します。かぎりなくリアルで臨場感あふれる画面を提供することができます。このため，画面更新の速さや画面の美しさを求めるゲームを中心に使われるようになりました。

DirectXにはゲーム作成に便利な機能が多く，なおかつハードウェアとアプリケーションの中間に位置するドライバなので，異なるハードウェア環境でも同じソフトがそのまま変更なしで動作するなど利点が多くあります。でも，プログラマとしてこれらを利用することを考えると，少しめんどうかもしれません。処理を動かす手順が決まっているので，それをまず始めに理解しなければなりません。またデバッグもたいへんです。ドライバを介しているとはいえ，直接ハードウェアを操作するケースが多いので，間違った操作をすると意外とあっけなくフリーズしたりします。そのため，Windows 2000での開発や2台のパソコンをシリアルケーブルでつないでリモートデバッグしている人が多いようです。

Table 1-3 ● DirectX 8bで提供する機能

名称	種別
DirectX Graphics	画像関連の機能を提供する。Direct3D，Direct3DXが含まれる。DirectDrawはDirectX 8で3D部分と統合され，DirectDrawそのものはDirectX 7インタフェイスで提供される
DirectX Audio	MIDIや音声データなどのオーディオ関連の機能を提供する。DirectSound，DirectSound3D，DirectMusicが含まれる
DirectPlay	Internet接続などゲーム間の通信機能を提供する
DirectInput	ジョイスティックなどの入力機能を提供する
DirectShow	DVD再生や動画に関する機能を提供する
DirectSetup	適切なバージョンのDirectXをインストールする機能を提供する

◆必ずしも使う必要はない

この本では，DirectX 8を基にした画像を取り扱うコンポーネントを用意しましたが，必ずしもすべてのゲームで使う必要はないと思います。

絵や動きのきれいさとゲームのおもしろさは，必ずしも一致するものではありません。ビットマップへのアクセスをちょっと速くしたいのなら，Win32APIのCreateDIBSectionを使う方法もあります。ゲームの種類や規模，「どこにおもしろさを求めるのか」といったポイントによって，使用するかどうかを選択してください。

Chapter

2

ゲームのアルゴリズムとデータ構造

おもしろいゲームを作るためには，ゲームがどのような仕組みになっているのかを理解しなければなりません。この章では，ゲームの仕組みとそれを実現する方法について，ジャンルごとに分けて解説していきます。

Section 1 アドベンチャーゲーム

アドベンチャーゲームは，プレイヤーみずからがシナリオを選択していき，自分だけのストーリを作ることができるゲームです。このゲームの仕組みは，ほかのジャンルのゲームでも多用されているので，知っておけばあとで役に立つことでしょう。

● あのときにこうしていれば……

人間を20年以上もやっていると，人生を左右する大きな選択がいくつかあったことに気がつきます。何人かの人たちに出会えていなければ，いまの私はないでしょう。逆に「あのときこうしていれば……」と思うこともしばしばです。現実では，過ぎたことはやり直すことができませんが，ゲームでは「リプレイ」すれば，そこをやり直して進むことができます。これをそのままおもしろいと取るか，リプレイできない本物の人生のほうが楽しいと取るかは，人それぞれです。

こうした「選択」により物語を進めていく「ロールプレイング」方式のゲームの1つが，「アドベンチャーゲーム」です。このアドベンチャーゲームを，本書で紹介するいちばん最初のゲームとして取りあげ，そのアルゴリズムとデータ構造を見ていくことにしましょう。

● アドベンチャーゲームとは？

アドベンチャーゲームほど応用が効くものはありません。本格的な小説のようなタイプから主人公を育てるようなゲームまで，幅広く作れます。ゲームシステムそのものも，ちょっと手を加えてやることで別の種類のゲームができます。パラメータやフラグを重視すると主人公を育てていくゲーム，主人公があちこちをさまよう部分だけ抜き出すと迷路の中を歩くようなダンジョンゲームになります。さらにこれを，画面を上から見た平面の視点にすれば，そのままロールプレイングゲームにもなります。これから紹介していくゲームたちにも，このアドベンチャーゲームのエッセンスが組み込まれているのがわかるはずです。

最近はいろいろなジャンルの違うゲームと組み合わされて，一概に「これはアドベンチャーゲームだ」とはいえないゲームが増えています。また昔ながらのアドベ

ンチャーゲームはいまでは少なくなってしまい，基本的な動きをつかむことすら難しいはずです。

◆選択の結果により物語が進んでいく

本来のアドベンチャーゲームとは，「選択肢をプレイヤーが選んで物語を進めていく」ゲームのことをいいます。ゲームの中で主人公が人生の岐路に立っているとき，プレイヤーがどちらへいくか，その道を指示することで，ゲーム上の物語が変わっていきます。

この選択の方法はゲームによっていろいろです。表示される設問の回答結果，鍵や品物などの「アイテム」，人と出会ったり何かが起きたときの「イベント」といったものによって主人公の進む道は左右されます。一部のゲームでは，ゲーム中に行われるまったく違うジャンルの「ミニゲーム」の得点や，「実際のゲームの流れとはまったく関係ない行動や場面が実は影響を与えている」といったことまでが物語の変更の条件となっています。

アドベンチャーゲームのおもしろさは「プレイヤーの意思で物語を変えられる」「主人公といっしょに擬似体験ができる」といったインタラクティブ的な要素にあります。これはほとんどのゲームの原点にもなっています。現在流行しているゲームが「体験」「シミュレーション」という共通のキーワードがあるところからも，これがわかります。

◆多彩なストーリと世界観

ストーリやゲームの世界観は，非常に多くの種類があります。J.R.R.トールキンの『指輪物語』から始まった「剣と魔法の世界」を題材にしたよくあるものから，「学校」「病院」といった1つの現実の環境をテーマにしたり，「戦争」「冒険」「歴史」はたまた「宇宙」といった広大なテーマを持つゲームもあります。映画やドラマ，小説などがジャンルやテーマに縛られることなく，多くの作品があることと同じです。もちろん本職の小説家が書いたシナリオがそのままゲーム化されたこともありました。最近では1つの作品を「テレビ」「雑誌」「CD」などで同時展開する「メディアミックス」がよく行われているので，ゲームのシナリオとなる原作が「マンガや小説として出版」「テレビで放映」されるということも多くなっています。この逆となるケースもよくあります。

アドベンチャーゲームの中身

ゲームの動きから基本的なアルゴリズムを考えてみます。Fig. 2A-1は典型的なアドベンチャーゲームの画面構成です。主人公が移動したり設問に答えたりすると，グラフィックや表示されているテキストなどが変わります。

この画面に表示されているデータを1つの「場面」(シーン)として考えます。そうするとアドベンチャーゲームというものは，場面が切り換わって進められていくことがわかります。これは映画や小説でも同じです。ただし，アドベンチャーゲームでは，次の場面をプレイヤーが選択することができます。これがアドベンチャーゲームのおもしろいところです。

グラフィック
テキスト
設問
●BGM
●効果音

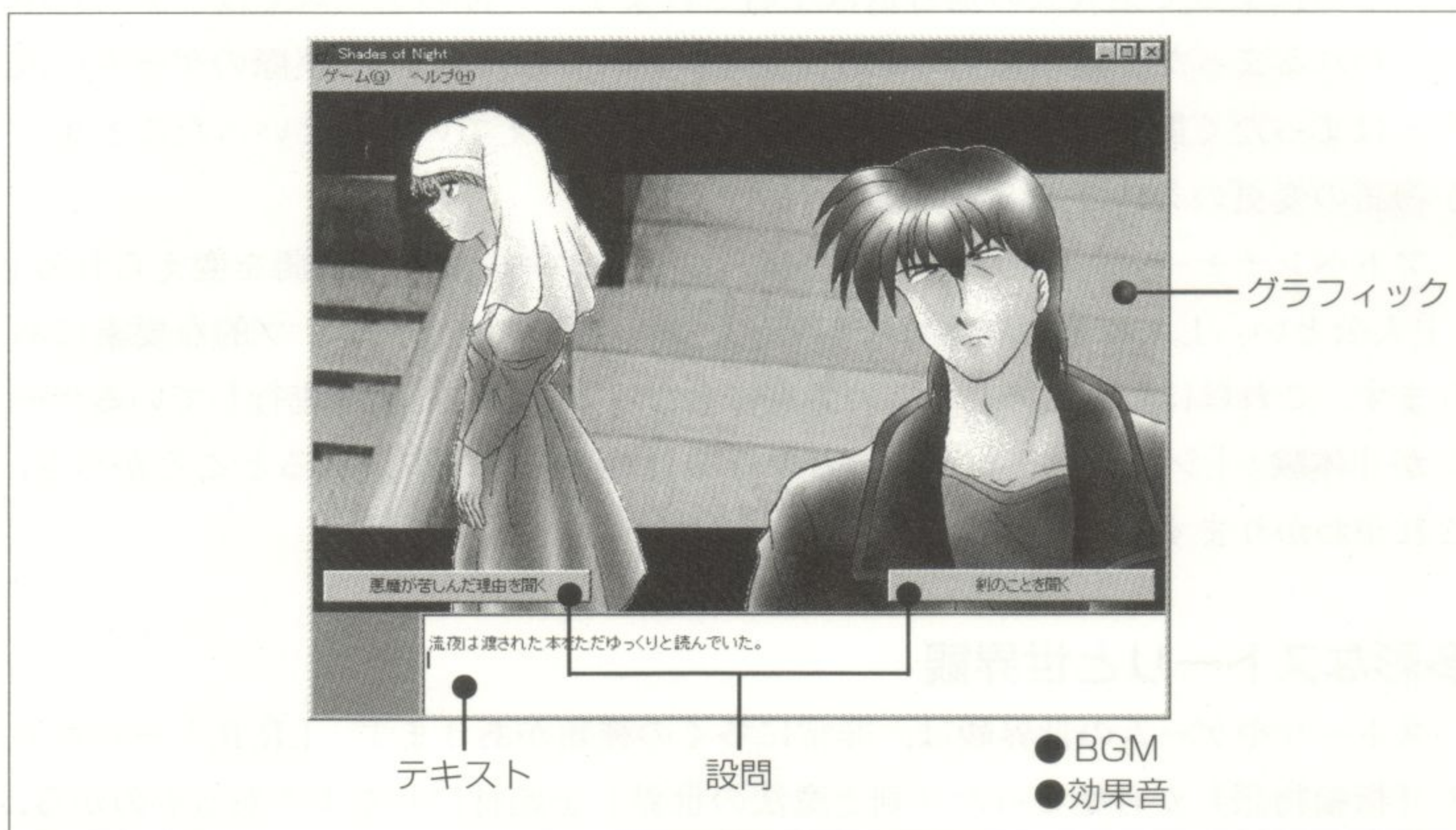



Fig. 2A-1 ●アドベンチャーゲームの画面。ゲームプレイ時にはBGMや効果音などが加わる

◆設問に従って次の場面へ移る

さて，ここで場面と設問の関係を見てみます。設問の数が少ないもの，たとえば「Yes」「No」で選ぶ二択の場合では，天秤がぶら下がったような形になります。図にするとFig. 2A-2のようになります。ゲームを進めていくと，必ずいちばん最後の段のどれかの場面，つまりエンディングにたどり着きます。なお，実際のゲームでは，クエストモードや複数の設問があるので，こんなにきれいに分岐されている形ではなく，もっと複雑な形になります。

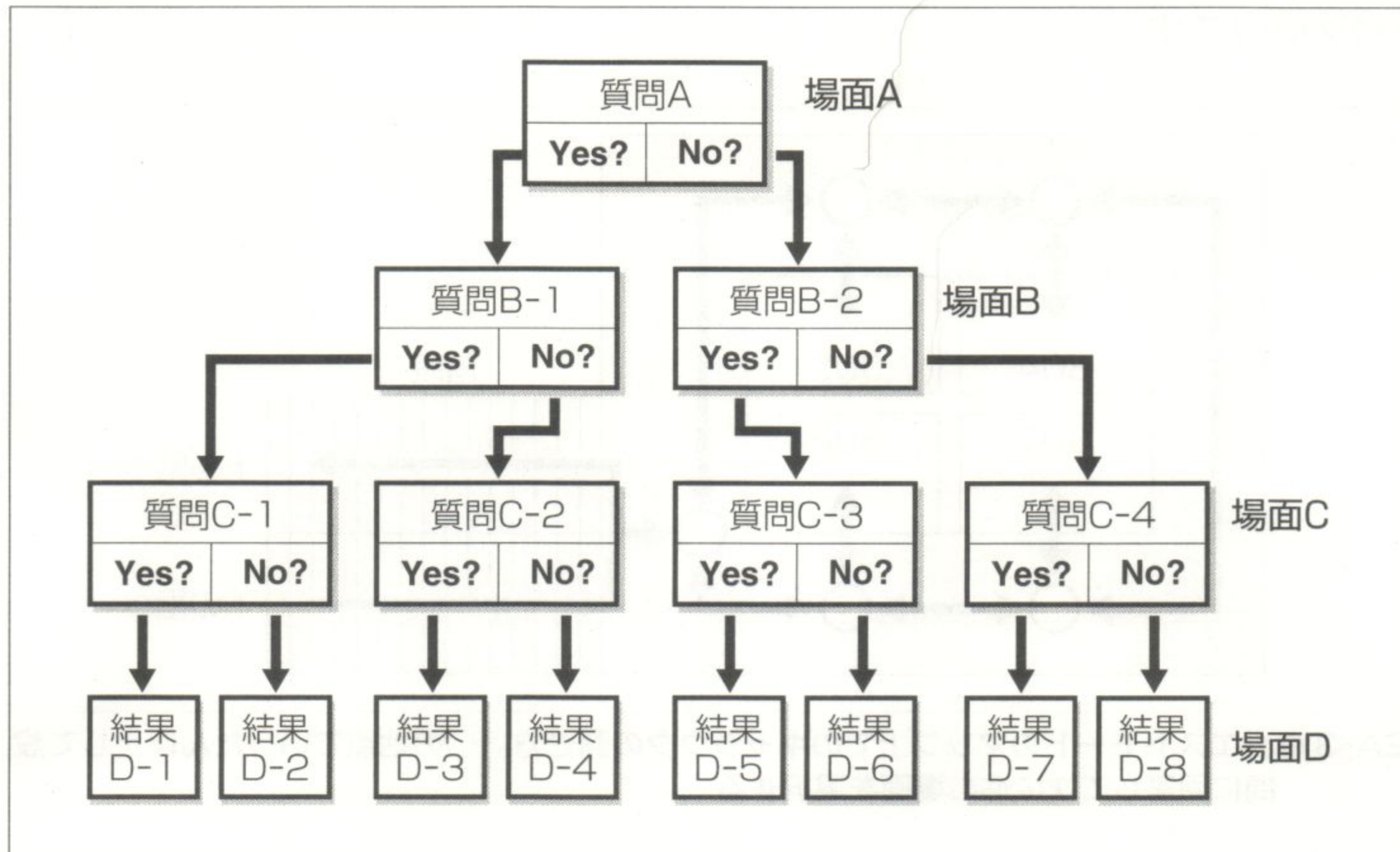



Fig. 2A-2 ●質問の答えによって次の質問が決まり，それと同時に場面も選択されていく

ゲームプレイ時には，場面ごとにプレイヤーに設問を示し，選択された回答に対応する次の場面を呼び出すようにしていきます。これを最後(エンディング)まで繰り返すことでゲームを進めていくのです。

アドベンチャーゲームでポイントになるのが，「場面」と「設問に答える」こととそれに従って「次の場面へ移る」ことです。これが動きの基本です。

◆クエストモード

アドベンチャーゲームではよく，部屋を探したり道を歩くという場面があります。これを「クエストモード」と呼んでいます。ただし，この名前については標準的な呼び方はなく，ゲームによっていろいろな呼び方がされていると思いますが，本書ではこのように呼びたいと思います。

クエストモードではゲームの進行が「分岐」していくのではなく，どちらかというと「ぐるぐる回る」ようなイメージになります。たとえばビルのある階で部屋を探すといった場面では，プレイヤーはFig. 2A-3のように動くことになります。「○」はいったん立ち止まってからほかの移動先へ移る地点を示しています。立ち止まる場所が部屋の前なら「この部屋を調べますか？」といった質問がされるところです。こうしたものを「移動先を聞く場面」と「そこに移る動き」として図にまとめると，Fig. 2A-4のような，複数の移動先が連結された複雑な形になっている

のがわかります。

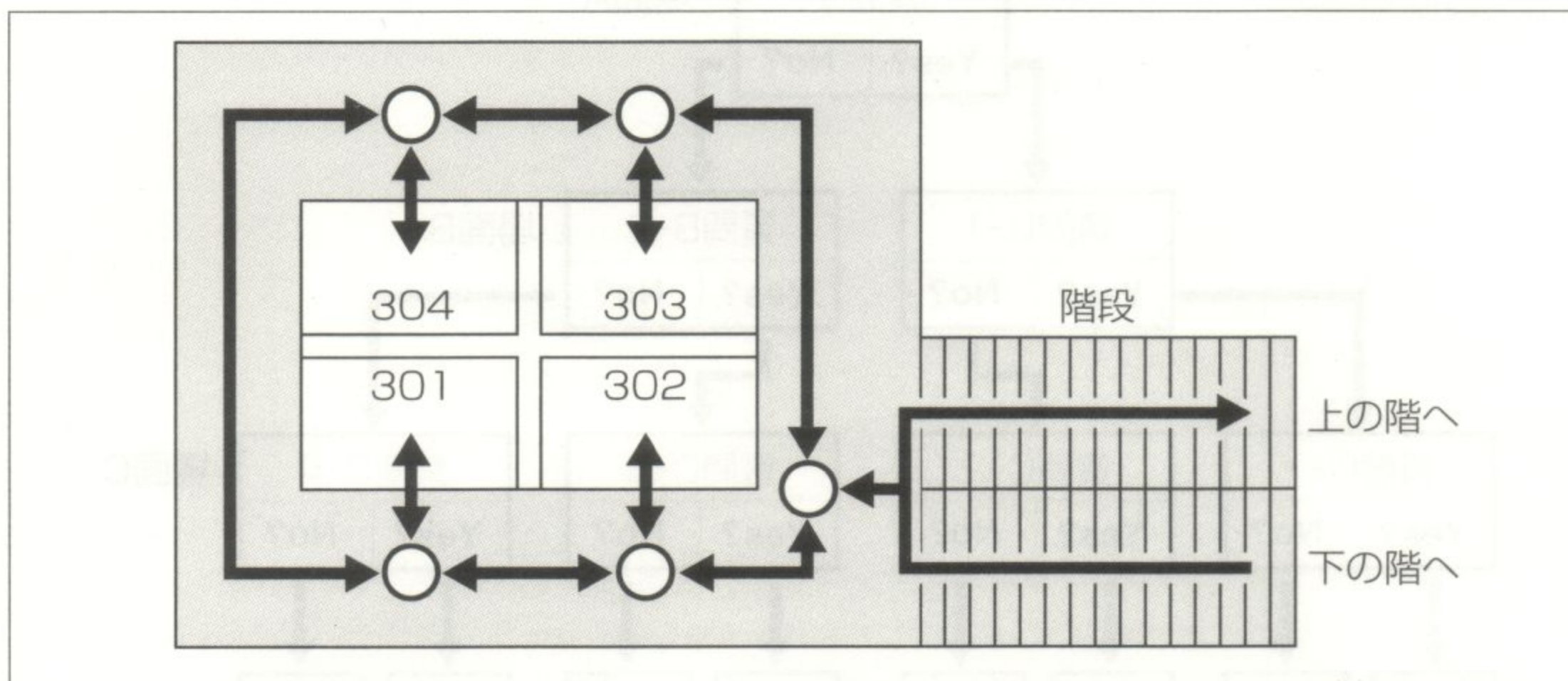



Fig. 2A-3●クエストモードのマップ上でのキャラクタの動き方。○の地点でいったん停止して設問に回答して次に進む場面を選択する

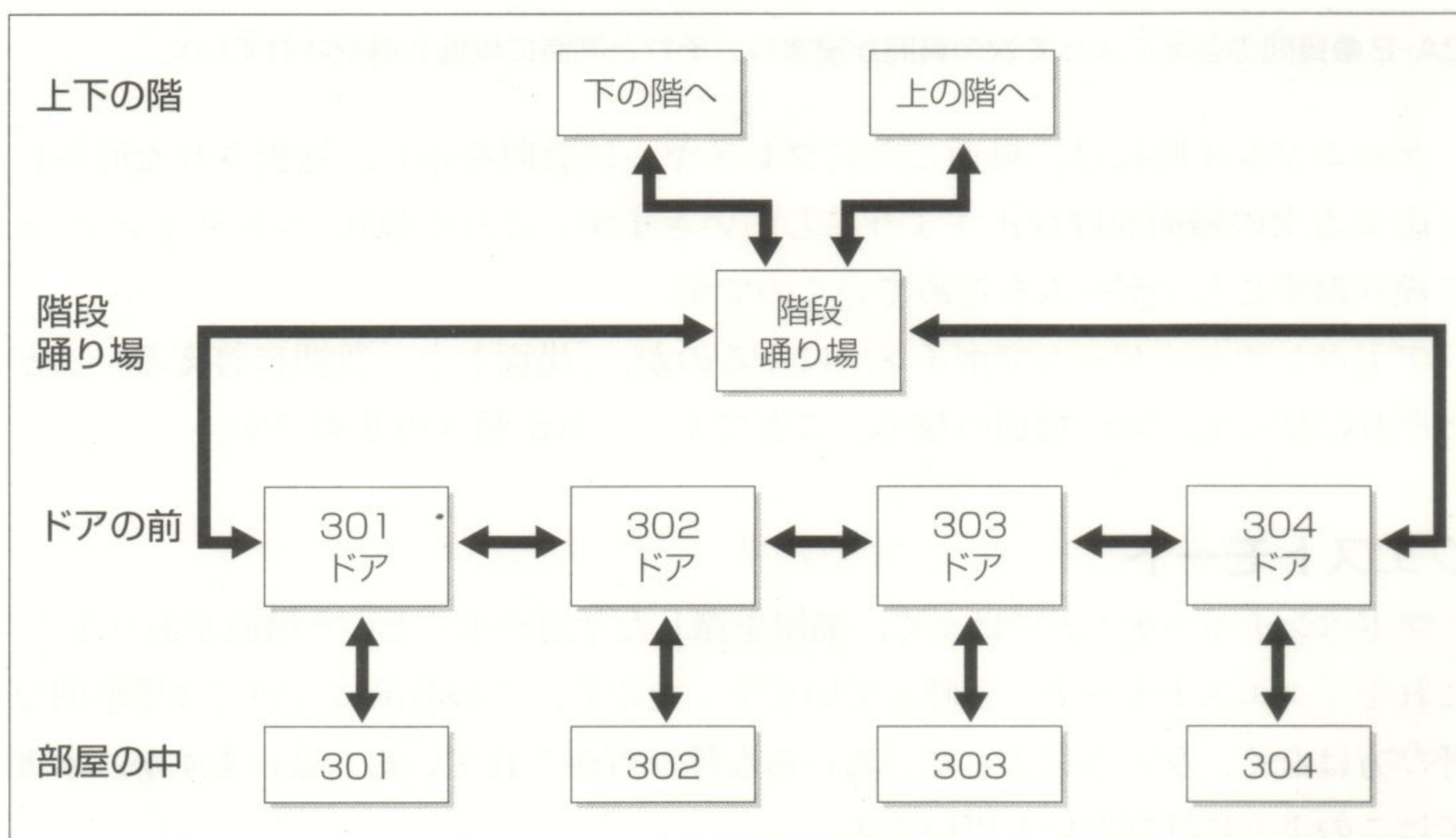



Fig. 2A-4●クエストモードでのキャラクタの移動先と移動経路

◆「イベント」と「フラグ」

ある場面では，ほかの場面とは違った特別な動作が必要なことがあります。たとえばほかの部屋にはない鍵が，ある特定の部屋にだけは落ちていたりします。この特別な動作を「イベント」といいます。イベントは次に解説する「フラグ」とともに，アドベンチャーゲームにはなくてはならない機能の1つです。

ある部屋で鍵を見つけると，いままで開けられなかった扉を開くことができ部屋に入れるようになるかもしれません。この鍵はプログラム上では1つの変数となっています。鍵が必要な場面でその変数を調べることで，鍵のかかった扉を開くというイベントを行うかどうか判断します。また鍵を取得することができる部屋にプレイヤーが入ると，この変数を「鍵をプレイヤーが持っていることを示す」状態にします。こうした動きをする変数のことを「フラグ」と呼びます。

ほかのジャンルのゲームでもそうですが，このフラグ管理がよくないといろいろなバグが発生します。もっとも致命的なのは「物語が先に進まなくなり，ゲームが終わらない」というバグです。これは必ず避けなければなりません。

● シナリオのデータ構造

アドベンチャーゲームのプログラムでは，各種データや設問を選んで入力する処理などが必要です。もし，もっとも単純に作るとしたら，場面を表示する関数を設問に従って実行するような形になるでしょう。ただし，これでは前の場面に戻ったり，ほかの場面に飛ぶといったときの処理が非常にめんどうになります。

◆データを収める枠を用意する

そこで，1つの場面に1つの「枠」を与えるようにします。この枠の中に次の枠を示す「道」(ポインタ)を付けてやります。そうすると，その道をたどることで次の枠へ移ることができます。移った枠からはその枠に対応する場面を取り出すことができます(Fig. 2A-5)。こうした仕組みを使うことで，場面の移動が簡単に実現できます。複数に分岐させたいときは，道を増やすだけで済みます。クエストモードでは，それぞれの道を枠の間で張り巡らせばいいのです。

「枠」はデータを収める「うつわ」にもなります。この「枠」にその場面で必要なデータをすべて収めます。たとえば場面の状態を示すグラフィックや物語のテキストデータなどです。これらはポインタの形にして枠に格納します。また設問に使うテキストも，枠の中に入れておきます。これ以外にも1つの場面で利用するデータを追加したいときは，枠を拡張するようにして追加します。

◆枠は構造体にして処理する

この枠を利用してプログラムを作るにはどうしたらいいのでしょうか？ C/C++やDelphiでは，枠を「構造体」として作ることができます。これはChapter1で解

Fig. 2A-5●その場面で必要となるデータとポインタを枠に入れて管理する。場面の移動は枠内のポインタの値に従って，次の場面のデータを収めた枠に移動する

説した「連鎖するデータ」をちょっと応用するだけです。

枠への道が複数になるときは，構造体を指すポインタを配列の形でテーブルにします。条件分岐文を省くため，設問の答えに番号を指定しておき，選択された解答の番号に対応するようにテーブルの中身を決めておきます。

List 2A-1，2A-2(P57)は枠を構造体として定義した例です。あとは変数を宣言するときにこれを型として使います。静的なデータとして構造体を連鎖させるときは，エンディングから最初のほうへ，つまり後ろから定義してから，ポインタ演算子を利用して移動先を加えます。

この連結された枠全体の集まりが「シナリオデータ」というデータ構造になります。

●イベント実行のアルゴリズム

「枠」の中には，その場面を構成するあらゆるデータを詰めました。それではイベントも枠の中に含めることができるでしょうか？

イベントにはいろいろな種類があります。もしイベントが枠に収めることができるデータなら，枠を拡張するだけで済みます。ですが，データ以外のプログラム処理が必要なら，このために特別の関数を用意することが必要です。

このイベントで必要な処理に合わせて，関数の形を決めます。関数の中でデータの処理をしたいときは，引数としてデータを渡します。これ以外にもほかのイベントと共通して必要なデータは引数として渡します。またフラグによる制御が必要なときは，この関数にもフラグを渡したほうがよいでしょう。返り値のほうは必要に応じて返します。

あとはこのイベント用関数を枠に収めるだけです。Chapter1で解説したように，「関数もポインタにできる」ことがわかっています。そこでこの方法を利用して，先ほどの「枠」へイベントを処理する関数をポインタとして収めます。イベントの実行は，この枠に収められているデータを処理するときに行います。

● フラグの管理

フラグに関係するイベントを実行するときは，そのイベントの中でフラグの状態をチェックするのが普通です。もっともめんどうなのが各イベントをコントロールするフラグの管理方法です。方法としては「一元的に集中管理する」と「分散して各枠の中に収める」の2つがあります(Fig. 2A-6)。

「一元的に集中管理する」方法は，フラグをテーブルとして管理します。メモリなどの資源をムダなく使える反面，もし誤った操作をしたときには，ほかのフラグにも影響が出ることがあります。

「分散して各枠の中に収める」方法では，フラグも枠の中のデータにします。枠

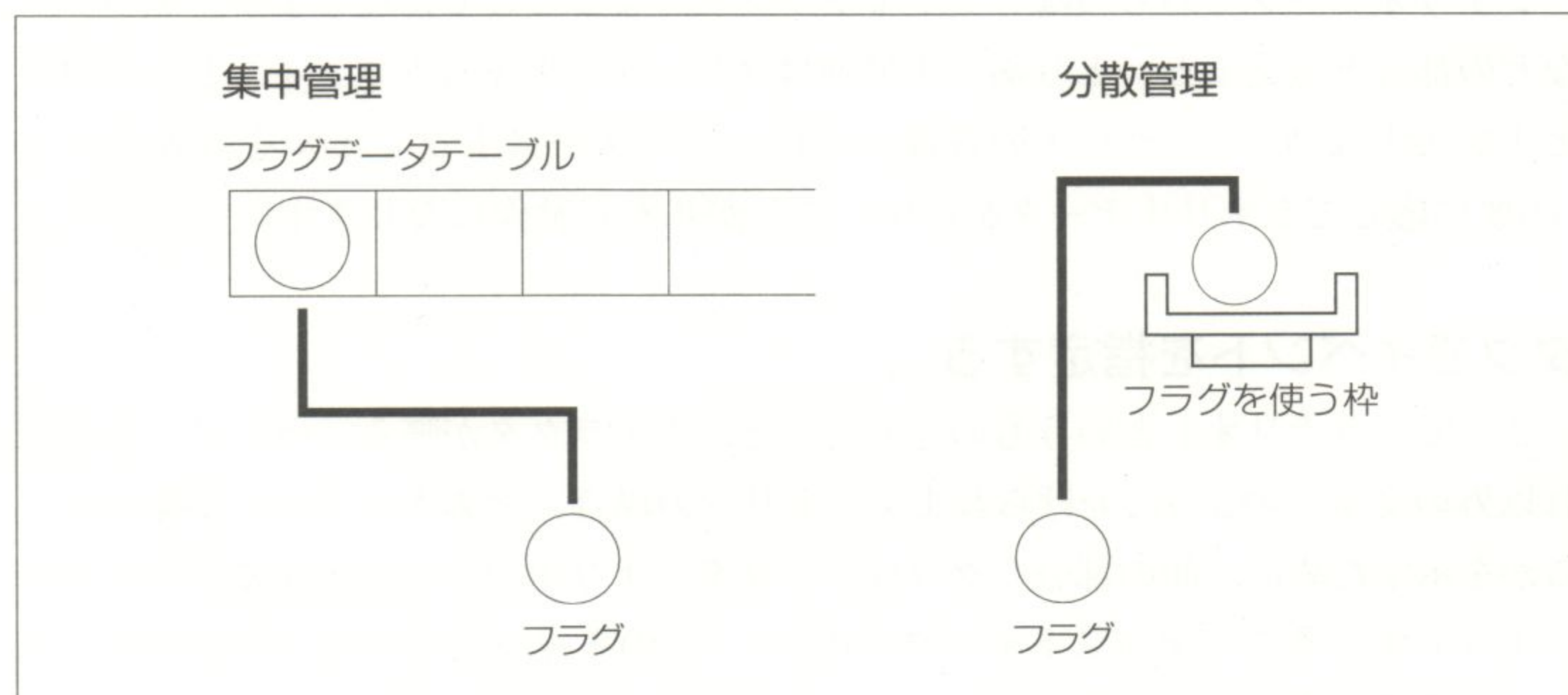



Fig. 2A-6 ●集中管理する方法ではフラグを１つのテーブルにまとめ，テーブルからフラグを引き出して操作する。分散管理ではフラグも枠の中にデータとして収め，その枠が選択されたときに同時にフラグも操作する

を選択することでフラグも選択できるので，特別な操作を必要としません。ほかの枠のフラグを操作したいときは，枠の中にそのフラグが収まっている枠のポインタを入れておきます。このポインタを介してフラグを操作します。

どちらの方法も一長一短があります。プログラムの規模と用途に合わせて使い分けるとよいでしょう。また「枠の中にテーブルにしたフラグの位置を入れておく」というように2つを組み合わせる方法もあります。

◆フラグ操作のミスを少なくするには？

誤ったフラグ操作を行わないようにするにはどうしたらいいでしょうか？　まず「似たようなフラグを複数作らず1つにまとめる」「フラグ操作を一元化する」ことが必要です。とにかくまぎらわしいことを避けるようにします。

また，フラグの操作では必ず「操作する」ことと「フラグの状態を見る」ことの2つの動作が組み合わされています。これに注目して，できるだけばらばらに操作するのではなく，枠が変更されたことに合わせて連動して操作できる形にするとよいでしょう。

● シナリオ実行のアルゴリズム

基本的に「シナリオを管理するもの」と「シナリオを表示するもの」は別にして扱います。

シナリオデータそのものはファイルやリソースなどとして管理します。デバッグなどの都合を考えると，とりあえず最初はプレーンなテキストファイルとしておくとよいでしょう。「シナリオの管理」では，このテキストファイルを読み込んで「必要に応じてシナリオデータを返す」ことが基本の動作になります。

◆タグでイベントを指定する

ここで「シナリオ」というものを考えると，キャラクタが喋る「セリフ」と「それ以外の文章」の2つに分けられます。セリフの場合，どのキャラクタが喋っているかを示すために，顔の部分のグラフィックをシナリオの横に表示するゲームが多いようです。そこでテキストデータの中に，「どのキャラクタのセリフなのか」をわかるようにするために記号の「タグ」を置くようにします。Fig. 2A-7はこのタグを使ったテキストデータの例です。

このデータのタグは，

```
＊＊mist_start1-1
私，ミストっていうの！
るんるんっ！きゃぴっ！

＊＊mist_start1-2
…ってノリ悪いわねえ。
へ，変なの～，きゃははは～
なに，君，それ～

＊＊demon_BatlMonster-1
ねえねえ，あれ見た？
やっぱりあの男の子かあいいよねー
コピー誌作ろうかな，なんてと思ったの～

＊＊demon_BatlMonster-2
ごめーん，これから即売会なの～
だから，早く倒れてよね。
```

Fig. 2A-7●シナリオデータのサンプル。誰がどの場面で喋っているかがわかるようにセリフの上にタグが設定してある

＊＊キャラクタ名_シーン一セリフの番号

となっています。タグは，わかりやすいものであればどんなものでもよいでしょう。特定のイベントや音声データの再生，フラグの操作なども，このタグを使えばシーンごとに確実に指定できます。タグを応用して，さらにシナリオの連鎖関係もこのテキストファイル中に入れるようにすれば，プログラム側のデバッグ作業が少なくなり，結果としてゲーム作りが簡単になります。

◆シナリオデータの実行

シナリオデータを実行するには，そのための専用関数を使います。これは前述した「シナリオを表示するもの」に属します。この関数では「シナリオが終端(エンディング)になるまでポインタを移動する」ことと「そのシナリオの場面を解釈して実行する」ことを仕事にします(Fig. 2A-8)。List 2A-3，2A-4(P58)は，これを基にしてコーディングした例です。

ポインタの移動にはポインタ変数を1つ用意します。設問にプレイヤーが返答して移動先の場面が決まったら，ポインタ変数に次の移動先を示すポインタの値を入れます。これを終わりになるまで繰り返していきます。各場面のシナリオデータは用意したこのポインタ変数から取り出します。

場面ごとの処理では，「枠」に定義されたデータを全部処理します。グラフィッ

クやテキストなら画面に表示してやり，イベント用の関数があるのなら，その関数を実行します。

次に設問を表示して，プレイヤーから入力する関数を用意します。引数は枠の中にある設問に必要なテキストや設問の数を渡します。返り値は選択された設問の番号です。この番号に従って次の枠へ移動するポインタをテーブルから引き出します。Windowsならこの部分をイベントとして考え，ここから上の部分の「プレイヤーに選択してもらうことを準備する」ものとその下の「プレイヤーに選択してもらった結果を実行する」部分は分けることになります。

設問がいらない場面では，設問用のテキストをなくして，かわりに0(NILまたはNULL)を入れることにします。これをチェックすることで設問処理用の関数を呼び出すかどうかを決めます。

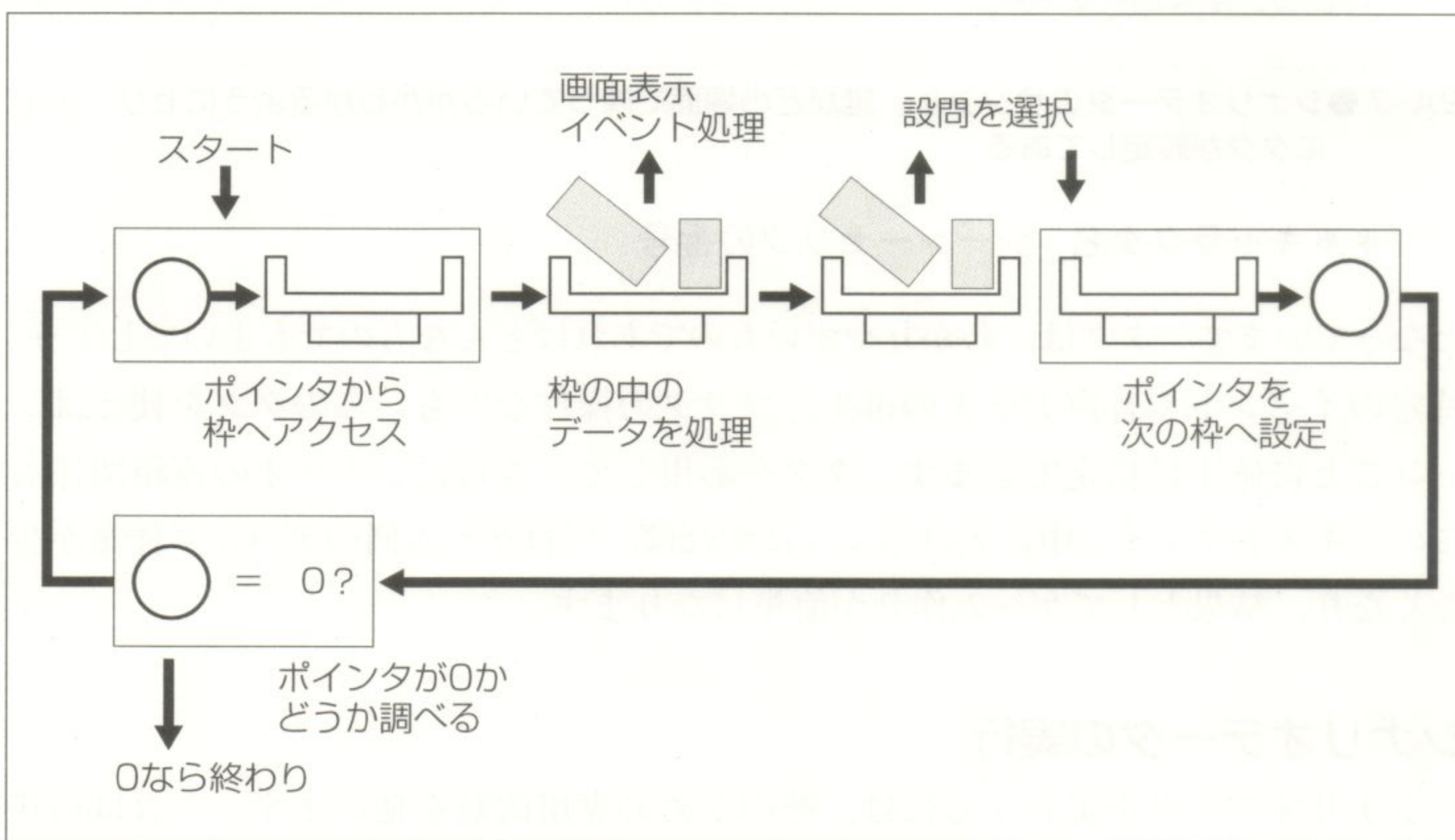



Fig. 2A-8●シナリオデータを実行する関数の内部構造

サンプルゲームの遊び方

実際にアドベンチャーゲームを作ってみました。このゲームでは，シーンごとのグラフィックが表示されます。エンディングは3種類あります。これもどちらかというと不思議な雰囲気を味わってもらうような形に仕上げました。暗い話が好きなもので，冒頭からかなり陰惨としてます(笑)。

グラフィック表示部分にマウスカーソルを置き，クリックボタンを押し続けると，

メッセージを早送りできます。また，背景は現実の風景をデジタルカメラで撮影したものを加工して使っています。

今回は作りませんでしたが「シナリオコンパイラ」というのものも，ある程度の大きさのゲームを作るなら必要になります。これはテキストやグラフィックデータからシナリオデータのファイルを作り出すものです。ゲーム本体のプログラムでは，音声データをプレイヤー側のソフトで再生するのと同じように，ファイル化されたシナリオデータを読み込むことでゲームを開始します。こうしてデータとプログラムを分離しておけば，同様のゲームを新たに作るときにかなり楽をすることができます。

こうしたアドベンチャーゲームの原理は，ハイパーテキストやWindowsのヘルプファイルでも基本的には同じです。こうしたものを利用してゲームを作るのもおもしろいと思います。

List 2A-1 ●枠を構造体にする（Delphi）

```
{ 型宣言 }
type
    wakup = ^waku;                        { 道（ポインタ）の型     }
    waku = record                         { 枠の定義              }
        nextp : array[0..4] of wakup;     { 複数の道（ポインタ）   }
        text : ^string;                   { 物語のテキスト         }
        que_text : ^string;               { 設問文のテキスト       }
        que_n : integer;                  { 設問の数              }
        gra : ^string;                    { グラフィックデータ     }
    end;

{ 注）グラフィックデータなどはメモリへのポインタという形よりも
    そのグラフィックデータのファイル名やメモリ上で何番目に収
    めたかという番号といった形で収める
}
```

List 2A-2 ●枠を構造体にする（C/C++）

```
/* 型宣言 */
typedef struct waku_t {            /* 枠の定義            */
    struct waku_t *nextp[3 + 1];   /* 複数の道（ポインタ） */
    char *text;                    /* 物語のテキスト       */
    char *que_text;                /* 設問文のテキスト     */
    int  que_n;                    /* 設問の数            */
    char *gra;                     /* グラフィックデータ   */
};
```

List 2A-3●シナリオデータの実行(Delphi)

```
{ 関数本体 }
procedure scenario_exec(temp_wakup : wakup);
var
   i : integer;
begin
   while temp_wakup <> nil do begin        { ポインタが nil なら終了    }
adv_print(temp_wakup);                     { グラフィックやテキストを表示 }
      adv_eventexec(temp_wakup);           { イベントの実行              }
      i := adv_isnext(temp_wakup);         { 設問を表示して移動先を取得   }
      temp_wakup := temp_wakup.nextp[i];   { 次の構造体を設定            }
   end;
end;
```

List 2A-4●シナリオデータの実行(C/C++)

```
/* 関数本体 */
void scenario_exec(waku_t *tempwaku)
{
   int i;

   while (tempwaku != NULL) {                   /* ポインタが NuLL なら終了    */
      adv_print(temp_wakup);                    /* グラフィックやテキストを表示 */
      adv_eventexec(temp_wakup);                /* イベントの実行              */
      i = adv_isnext(temp_wakup);               /* 設問を表示して移動先を取得   */
      temp_wakup = temp_wakup.nextp[i];/* 次の構造体を設定            */
   }
}
```

Section ② ロールプレイングゲーム

ゲームの花形ともいえるのがRPG（Role Playing Game＝ロールプレイングゲーム）です。このゲームにはさまざまな技術が複雑に組み合わされています。それらに使われているアルゴリズムとデータ構造の中からマップとキャラクタのデータ管理や処理方法について見ていくことにしましょう。

● 小さな部品を組み合わせる

ジグソーパズルで遊んだことがありますか？　1枚の写真や絵を小さな破片に切り分けて，それを再び元の姿にしていくだけの単純な遊びです。でも単純なだけに思わず熱中してしまうおもしろさがあるようです。私は以前，パズル雑誌の懸賞としてジグソーパズルをもらったことがあります。「なぜパズルを解いてまたパズルをもらわねばならんのだー」と思いつつ，3日かかって完成させたりしていました。

世の中を見回すと，ジグソーパズルのように「小さな部品を組み合わせて1つのものを作る」という方法はとても多く使われています。平面だけではありません。最近のビルや家では，工場で部屋などを1つのユニットとして作って，まるでブロックでお城を作るようにそれを積み上げていく工法もあります。

実はゲームの中でもこの「小さいパーツに切り分けて組み立てる」方法が画面表示などに取り入れられています。ここで取りあげるRPGというゲームでは，とくにマップとキャラクタ部分にこの方法が使われています。これを中心にして解説したいと思います。

● ロールプレイングゲームとは？

まずRPGとはどういうものだったか，振り返ってみることにしましょう。RPGはタイトルこそ多数ありますが，内容的にはもっとも固定化されたゲームの分野かもしれません。現在RPGといえば「キャラクタがゲームの世界を移動して物語を進めていく」ゲームのことになります。

◆「キャラクタ」と「マップ」

登場人物などを指して「キャラクタ」といいます。たとえばあるサッカーチームにいるキャラクタは「野人岡野」ということができます。キャラクタという言葉は，ゲームに登場するものの「総称」としても「1つのものを指す言葉」としても使われます。RPGでは「キャラクタ」はその世界の「マップ(地図)」の上に投影されます。プレイヤーはマップの上に載せられたキャラクタをゲームの画面として見ることになります。ゲーム画面の視点は地図の上から見たものがほとんどです。そして最初は世界全体が見えないようになっています。プレイヤーが操作する主人公のキャラクタがマップの上を歩くことにより，世界の全体像がわかるようになっています。この謎解きの要素もRPGのおもしろさのポイントの1つです。

RPGの画面に表示されるマップの視点を変えれば，そのままダンジョン(迷路)ゲームに作り変えることもできます。RPGの視点は「地図を上から見た」ものですが，ダンジョンゲームでは「キャラクタが実際に見ている」視点になります。キャラクタが前に歩けば壁が迫ってくるように見えるのがダンジョン系ゲームの特徴です。プログラム上ではグラフィック表示の変更を行うだけで，比較的簡単に作ることができます。このように，RPGとダンジョンゲームは密接なかかわりを持っています。また視点が「キャラクタが横や縦方向に進むのを見ている」ことになると「アクションRPG」などと呼ばれる分野のゲームにもなります。

◆世界を旅することでゲームが進んでいく

ゲームの進行は「世界を旅する」ことで進んでいきます。与えられた目的を果たすために，ユーザがキャラクタを動かしてゲームの中の世界を歩き回ります。その途中で敵と遭遇したり，町を訪れて必要な装備の買い物をするなどのイベントが起こります。ときにはほかの仲間を引き連れ「パーティ」となることもあるでしょう。このイベントの展開はほかのジャンルのゲームと比べてもかなり自由度があり，それを特徴にしているRPGもあります。

物語の内容は「主人公を敵と闘わせて経験値を高めたあとに敵のボスと対決する」といった「キャラクタを育てて敵を倒す」ものが王道かもしれません。世界観の設定としてよく使われるものとして「剣と魔法の世界」というものがあります。中世以前の時代設定で剣士や魔法使いが魔物たちを退治するといったものです。こうした世界観そのものは前述の『指輪物語』などの小説から生まれたものですが，その世界を膨らませて発展させたのはテーブルゲームやRPGの力がとても大きいようです。

ここで「なーんだ，アドベンチャーゲームとどこが違うんだ」といわれる聡明な方もいるでしょう。RPGは大きく見ればアドベンチャーゲームと同じく，シナリオを進めていく種類に分類されるゲームです。でも，どのゲームとも違う大きな特徴があります。それは「自由に遊べる世界そのものを提供する」ことです。私たちがどこかへ旅をすれば予定外のハプニングが起きることがあります。最近のRPGでは本筋の物語以外にも，こうした実際的なハプニングや遊びを提供することが多くなりました。これはアドベンチャーゲームと違い「世界の中を容易に動くことができる」からこそより楽しめるのです。

RPGは，ドキドキハラハラしながら世界を冒険して物語を解明していくおもしろさ，敵を倒す爽快さ，そして自分のキャラクタを育てる楽しみなどを非常にうまく組み合わせることができるゲームです。この点をしっかりとプレイヤーに伝えることができたゲームがもっとも支持されているように思います。

● ロールプレイングゲームの中身

それではゲーム画面からどういう仕組みで動いているかを調べてみましょう。Fig. 2B-1はもっとも多く見られるタイプのRPGのプレイ中の画面です。キャラクタはマップ画面の中にいて，プレイヤーの操作により指定された方向へ歩きます。

●メインウィンドウ
マップ画面
パラメータ
キャラクタ
メッセージ
シナリオ表示
行動決定
●戦闘ウィンドウ
●BGM
●効果音

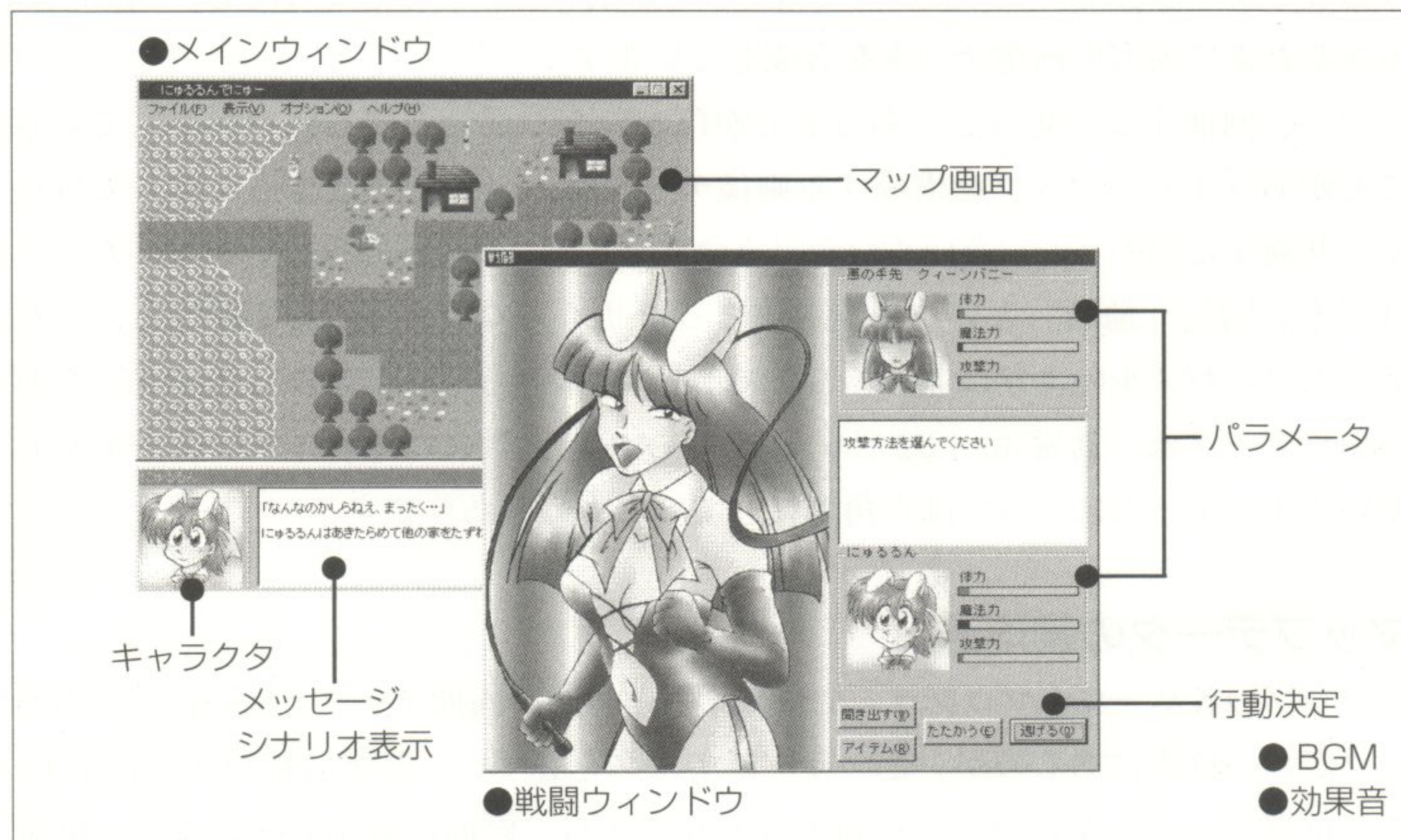



Fig. 2B-1 ● RPGの画面構成。ゲームプレイ時にはBGMと効果音が加わる

もしキャラクタが敵と遭遇するとマップ画面は敵との戦闘場面に切り換わり，敵と戦うことになります。敵を倒したり逃げることに成功すると再び元のマップ画面へ戻ります。マップ画面はこうしてイベントが起きるたび，にそのイベント用の画面へ切り換わります。

◆セリフやパラメータなども表示される

物語で必要なセリフなどは下のウィンドウ部分に表示されます。ゲームによっては右側のウィンドウにキャラクタの体力，魔力といったパラメータが表示されることもあります。

このほかにRPGでは「キャンプモード」というものがあります。キャラクタの移動とは関係ない特定のキーを押すことにより，セーブやロードといったシステム系の作業をできるようにしています。

RPGでの画面の動きはこのぐらいです。ゲームによっては画面構成など明確に違うこともありますが，基本的にはこうした動きに集約することができます。

● マップの処理

通常マップ画面に描かれる絵は「キャラクタ(人物)」と「マップ(背景)」に分けることができます。プログラムでもそれぞれ別のデータとして管理し，画面へ表示するときに両方の画像データを合成しています。

マップ画面をよく見ると，ある部分がほかのある部分と同じ色や形をしていることがあります。マップを構成する画像データは，「1つのまとまった絵」ではなく「共通する絵のパーツをいくつか組み合わせる」ことで作られています。たとえば「木」「岩」「地面」といった絵を決まったサイズのパーツにして保存し，これをジグソーパズルのように並べることで，1つの大きなマップになるのです。なおパーツデータは，通常32×32ドットといった8の倍数になるサイズで正方形をしています。ゲームによっては六角形にしているものもあります。

◆マップデータの管理法

マップの各パーツをプログラムで管理するときは，画像データとは別に「どのパーツがどの位置にあるのか」という位置データを必ずいっしょに持たせています。この位置データを基にパーツを組み合わせてから，画面に表示しています。位置データはジグソーパズルの切り込みに当たります。

これがマップのすべてです。なお最近のRPGでは，パーツ同士の類似点を見つけるのが困難なほどきれいな画面をしたものが多くなっています。これはいくつかの背景パーツを複数重ね合わせたり，色数やパーツ数そのものを増やすことで実現しています。また使えるメモリやグラフィック資源がよくなっているのも一因です。

● キャラクタの処理

ゲームの画面の中をプレイヤーの思う方向へ進む，物語の主人公をデフォルメしたかわいい奴，それが「キャラクタ」です。実際のゲーム画面を今度はキャラクタに注目して見ていくことにしましょう。

画面上でのキャラクタの大きさは，ほとんどの場合でマップを構成するパーツと同じぐらいの大きさです(Fig. 2B-2)。もしキャラクタが8頭身などで描かれると，とても細く小さく表示されてしまいます。そこでキャラクタの全身を上下に縮めた「デフォルメ」という作業が行われています。こうすることで，結果的に動きをはっきりさせることにもなっているようです。もっとも，視覚的よりもキャラクタのかわいさを強調している場合もあります。

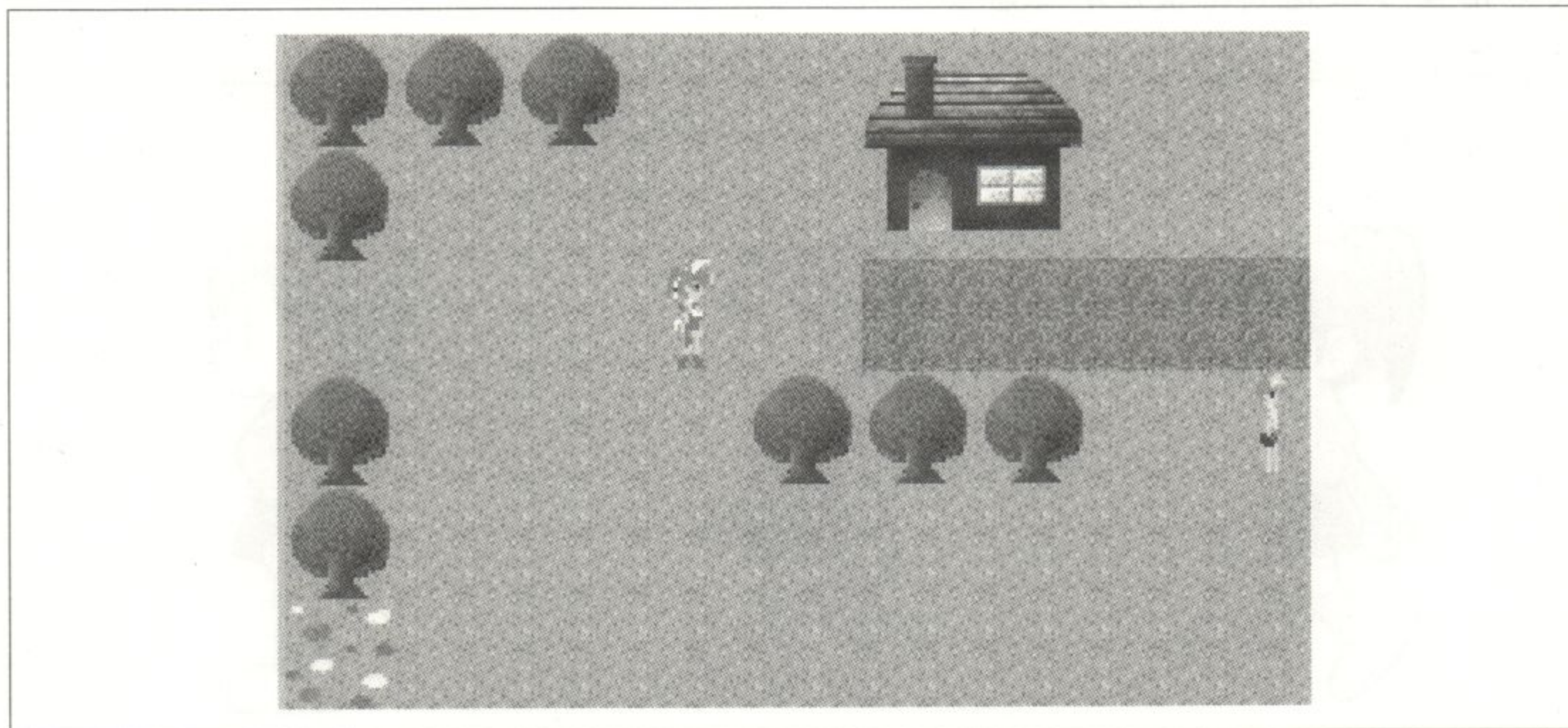



Fig. 2B-2 ●マップ画面上に表示されたキャラクタ

◆キャラクタの動かし方

マップでは「パーツを並べて画面を作る」のが基本でした。キャラクタが動いているように見せるには「あるパーツを連続して表示する」ことが基本になります。

RPGをプレイしている最中にキャラクタの動きをよく見ると，キャラクタが移動するときは，元いた位置にあるマップのパーツ上から，移動先にあるパーツ上へとキャラクタの絵を動かしているのがわかります。「1つのマップパーツ上にパーツと同じ大きさのキャラクタが描かれている」ともいえるでしょう。移動するときはいったんそのキャラクタを消してから，次のマップパーツの上に改めて別のパターンのキャラクタを描き加えています。こんな作業をずーっと繰り返しているだけです。

また同じ方向へ移動しているときには，必ず何枚かおきに同じパターンの絵が描かれています。「ある一連の絵を循環的に連続して表示している」のです。「リングバッファ」といったほうがピンとくる人がいるかもしれません。たとえば「1から順に3までパターンを描いたら，次のパターンは1に戻る」という感じです。こうして順繰りにパターンを描き続けることで，動いているように見せているわけです。だいたいのRPGではこの2つのポイントは，まず変わりません。

キャラクタを動かすときのアニメーションパターンは，もっとも簡単な場合で「上下左右」での4方向でそれぞれ「停止」「右足が前」「左足が前」の3パターンで構成することができます(Fig. 2B-3)。これらのデータをプレイヤーのキー操作に従って連続的に表示すれば，キャラクタがあたかもプレイヤーの手によって動いているように見せることができます。

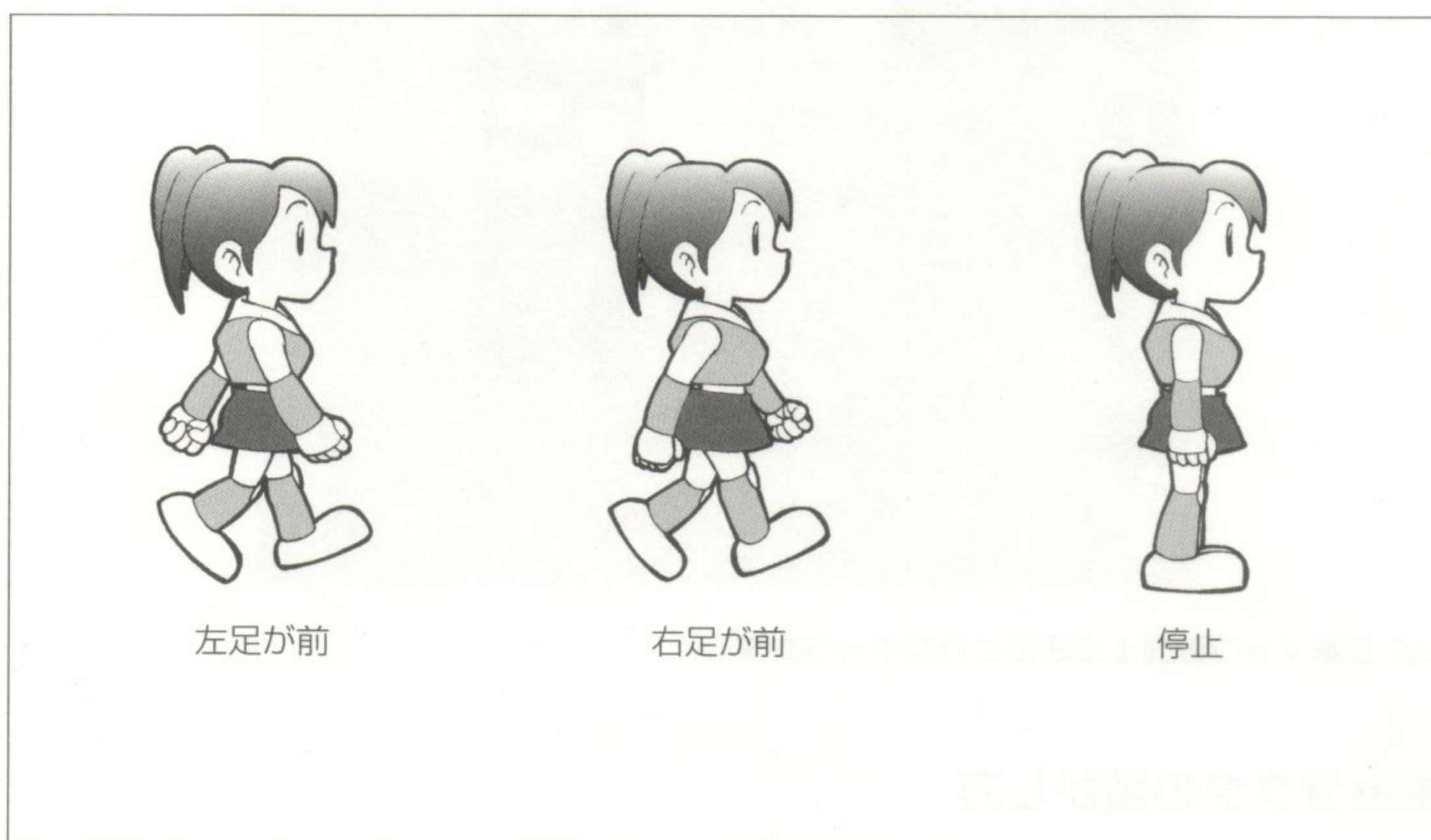



Fig. 2B-3●3パターンのグラフィックを用意し，このグラフィックの繰り返しによってアニメーションを行う

● マップのデータ構造

それでは，マップ画面にはどんなデータ構造が必要でしょうか？　マップのデータ構造の基本は「2次元配列」です。普通に作られた配列は横や縦に一直線の表としてイメージすることができます。これを「線形(リニア)配列」や「1次元配列」といいます。2次元配列ではこれを縦にも横にも参照できる状態にします(Fig. 2B-4)。2次元配列の宣言方法はTable 2B-1のとおりです。図を見ればわかるとおり，これだけでなんとなくマップ状態になっています。

◆セルに必要なデータを収める

この配列に「枠(セル)」を組み合わせてみましょう。マップの1つのマスを1つのセルとします。配列の並び方どおりに画像データを組み合わせていけば，マップ画面を作ることができるという処理にしてみます。

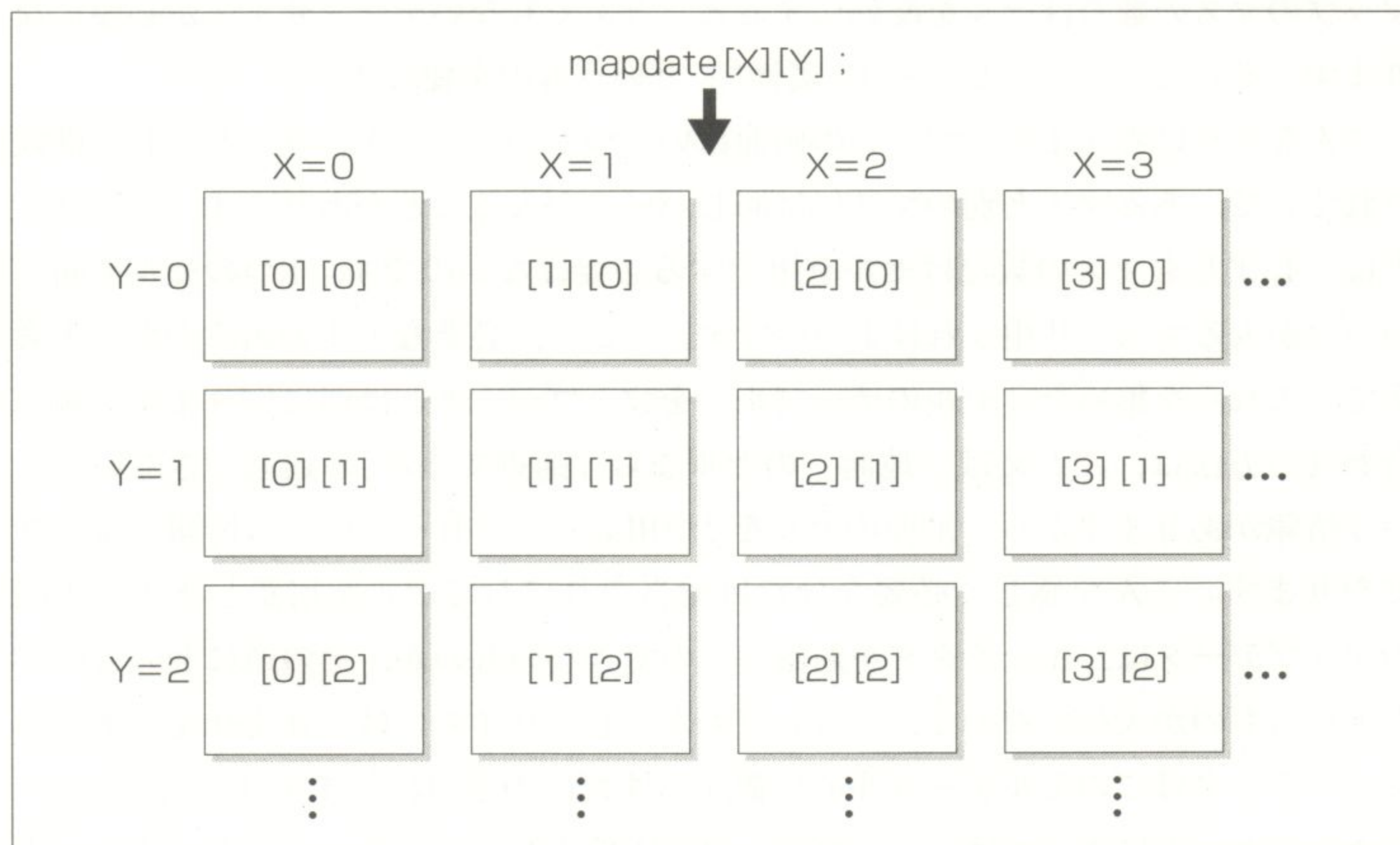



Fig. 2B-4 ● 2 次元配列の例。配列を参照するときには，X:Y の座標で指定する

Table 2B-1 ● 2 次元配列の宣言方法

Delphi (ObjactPascal)	C++Builder(C/C++)
配列の範囲を "," で区切って指定する 変数名 = array[値の範囲,値の範囲] of 型名; 例) data : array[0..64,0..64] of integer;	添え字を 2 つ重ねる 型名 変数名[値の範囲][値の範囲]; 例) int data[64 + 1][64 + 1];

まずはセルの中身です。マップの中の1地点を表すにはどんなデータが必要か考えます。その位置が何であるかを示す「画像」は必要です。これは画像データへのポインタとしてセルに格納します。キャラクタが通ることができるかどうか，またはどの方向へ移動できるのかといった「通過フラグ」やその地点にイベントがあるかどうかを示す「イベントフラグ」といったものも必要です(List 2B-1, 2B-2(P75))。

ここで考え方が2つ出てきます。「その地点に必要な情報を1つのセルとして管理」する方法と，「セルに収められた情報の種類ごとにマップを複数作る」方法です。後者は「レイヤ」と呼ばれる平面が上下に並んだ階層構造になっています。どちらも多く利用されている方法です。今回はセルにまとめて使ってみます。

◆アトリビュートでデータサイズを克服する

ここでよく問題になるのが，マップデータの「データサイズ」です。もし単純にマップのマスの数だけセルを配列にすると，とんでもないデータサイズが必要になります。そこで「アトリビュート(属性)」というものを使います。

こんなふうに考えます。マップの画面はいくつかのパーツが組み合わされて構成されていて，あるマスと別のマスには同じパーツが入ることがあります。ということは，必ずしもマスの数だけセルを用意する必要はないのです。1つのセルを同じパーツが入るマスで共用すればよいのです。そこで，必要なセルの組み合わせを考えて，これらを集めた「セルのテーブル」をマップデータとは別に作ります。組み合わせの方法は，たとえば「地面」のセルなら「通過できる」「通過できない」という情報がありますから，地面のセルを2つ用意してそれぞれにこの情報を持たせてやります。こんな感じで必要な分だけ総当たりにしてセルを用意します。実際のマップデータは，キャラクタが通ることのできないものか，その逆にどの方向でも進めるものが大部分なので，こうしてもたいしたサイズにはなりません。実際のマップデータはこのセルテーブルの位置を示すだけのものにします(Fig. 2B-5)。

さらにゲームによっては，マップデータの圧縮を行っていることもあります。方法としては「周りのセルとの関連や相関を取る」方法が多いようです。またキャラクタの動く方向を制限したり，4ビットでセルの種類を表すことやセルに入れる情報を減らしたりしてマップデータのサイズを減らしています。

データサイズとの闘いは，昔から変わらないプログラマ泣かせの点でもあります。

マップデータ
mapdate[0][0]
セルデータ
celltbl[0]
セル
celldata
その地点のデータ
グラフィック
通過フラグ
イベントフラグ

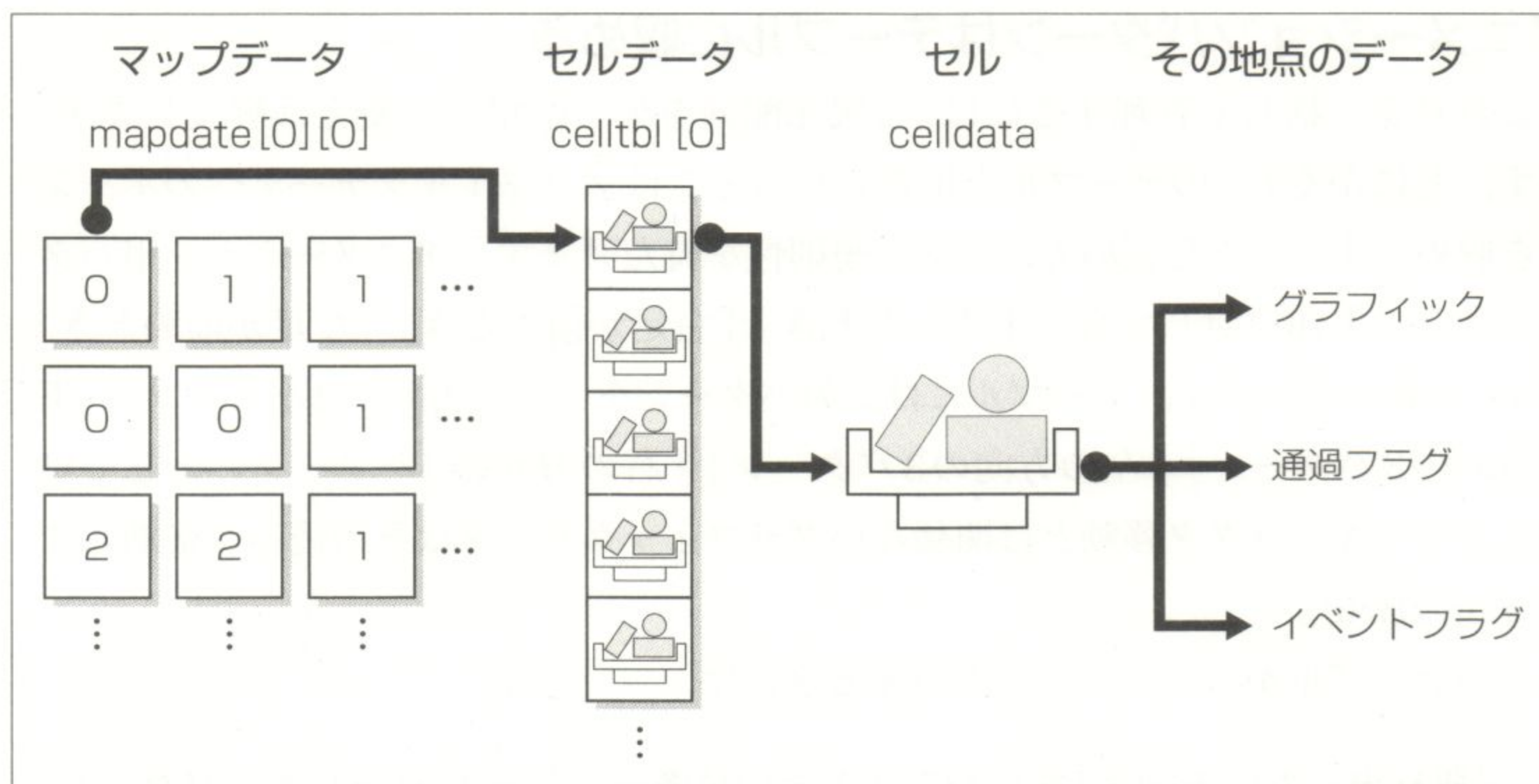



Fig. 2B-5 ●マップデータの値に従ってセルテーブルからセルを選び出し，そこから必要なデータを取り出す

◆マップ技術の応用

マップの技術はこれ以外にもとても広く使うことができます。RPGのマップでは，マップを作るための「マップエディタ」というものがあります。指定したエリアにパーツを選んで置いていくことで視覚的にマップを作ります。これをそのままゲームにすると「SimCity」タイプのシミュレーションゲームにもなります。マップとキャラクタの関係をそのままにしてパズル的要素を持たせたのがいわゆる「倉庫番」と呼ばれるゲームです。

● キャラクタのデータ構造

マップデータのセルを作ったときと同じように，キャラクタに関するデータをまとめた1つの構造体を作ることにしてみます。

アニメーションパターンは普通に格納するだけでは取り扱いがややめんどうになります。もしプレイヤーが「右へいく」と操作したら，それに関係するアニメーションパターンを一括して得られると便利です。そこで「パターン」と「方向指定」をうまく関連付けられるようにしなければなりません。テーブルにするならば「上下左右」の4方向が引数になっていて，それから一連のアニメーションパターンが得られるような感じです。でも，キャラクタにはこうしたデータ以外にも，戦闘シーンや顔などといったキャラクタと関連するグラフィックデータもあります。

◆アニメーションパターンはテーブルに収める

これらを一括して管理するには，2次元配列を少し応用して使えば解決します。まず，とにかく1つのテーブルを用意して，そこにグラフィックデータへのポインタを収めます。ただし収め方にはある規則性を持たせます。アニメーション用パターンはどの方向に対しても「止まったとき」「右足が前のとき」「左足が前のとき」の3つの順に並べます。テーブルにはこのパターンを「上の方向の3パターン」「下の方向の3パターン」「左の方向の3パターン」「右の方向の3パターン」として格納します。キャラクタ移動とは関係ないグラフィックデータはその後ろに格納します(Fig. 2B-6)。

このテーブルからデータを引き出すときには，

(取り出したい方向×(左足が前のときの位置＋１))＋取り出したいパターン

というようにします。もともと2次元配列では「X=640，Y=480のとき，X=32，Y=64は，64×640+32=40,992ピクセルの地点」というように「縦Yの地点は横Xの最大をY倍したところ」ということにして値を取り出しているので，これを少し変えて利用しているだけです。

アニメーションパターンではつぎつぎと絵を切り換えるために「いま描画した絵があるテーブルの位置」を保存する必要があります。移動するためにパターンを取り出すときは，前にグラフィックを取り出したテーブルの位置を保存するようにしておき，この変数を交互に切り換えることで，取り出すグラフィックも切り換えるようにします。

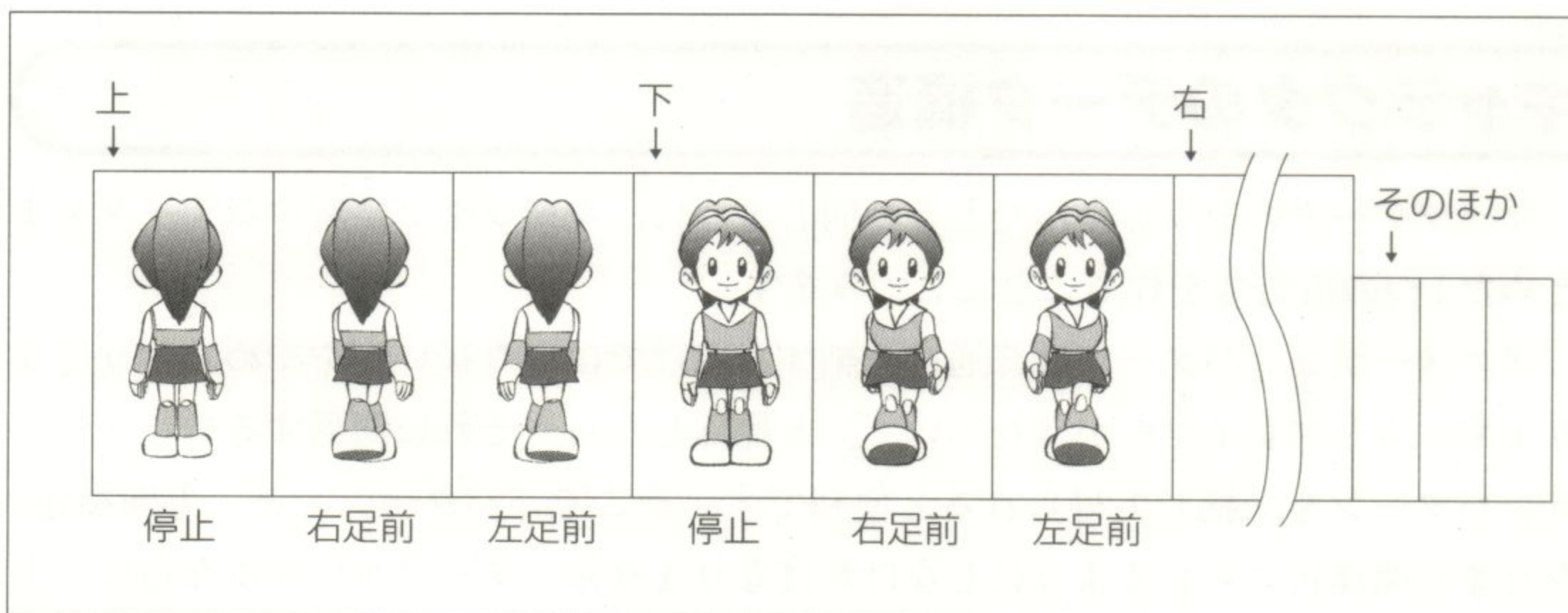



Fig. 2B-6●キャラクタデータのテーブル。テーブルの始めのほうに規則に従ってアニメーションパターンを収め，その後ろにそのほかのグラフィックデータを格納する

◆パラメータでキャラクタの状態を管理する

キャラクタにはグラフィックデータだけではなく，体力や魔力といったそのキャラクタに固有の「パラメータ」と呼ばれるデータもあります。モンスタを倒したりすると「レベルが上がった！」などと表示されますが，これはこのパラメータの値を増減しているだけです。ゲームプログラムでは，パラメータもキャラクタごとに管理しなければなりません。体力や攻撃力などはもちろんですが，こうしたプレイヤーの目に触れるもの以外に，ゲームシステムの内部で使う「システムパラメータ」というものもあります。モンスタといった敵の出現頻度やシナリオの進行具合，キャラクタが現在いる位置などです。これらはキャラクタごとに持たせる必要があるものだけをデータに含めるようにします。

なお，こうしたパラメータを管理する場合には，プレイヤーが途中でゲームを中断してゲームの進行状況を保存する「セーブ」という作業があることを考えておく必要があります。一括してファイルなどへ保存できるようにしておき，それが再び読み出されたとき，ゲーム進行へすぐに反映できるようにしなくてはいけません。とくにシナリオ上必要なフラグなどは個別に保存するのではなく，まとめた1つのデータとして持たせたほうが安全です。

こうしてできた構造体は(List 2B-3，2B-4(P76))，登場するキャラクタの分だけゲーム本体で定義して利用します(Table 2B-2)。こうするとソース上でも直感的にわかりやすいはずです。

● マップ表示のアルゴリズム

データ構造を決めたら，それを解釈するルーチンを作成します。作らなければならないのは「最初にマップを表示するルーチン」「ユーザからのキー操作を受け

Table 2B-2 ●キャラクタの定義方法

例：「岡野」というキャラクタを作りたいときは？

Delphi(Object Pascal)	C++Builder(C/C++)
定義方法	
var 　Okano: TCharacter;	character_t Okano;
参照方法	
Okano.Name := '野人岡野'； Okano.Hp := 200;	strcpy(Okano.Name, "野人岡野"); Okano.Hp = 200;

たときにキャラクタがその方向へ移動できるかを調べるルーチン」「マップをスクロールさせるルーチン」といったものです。

◆マップを表示する

マップデータを表示することそのものの処理はけっこう簡単に作れます(List 2B-5, 2B-6(P77))。マップデータを基に画像データを画面バッファに組み立てて表示するだけです。RPGではマップ全体を表示することよりも,「左上が20, 20の地点から30, 30サイズ分表示する」といった指定方法が要求されます。これらの座標データは構造体としてまとめてやり取りします(Fig. 2B-7)。

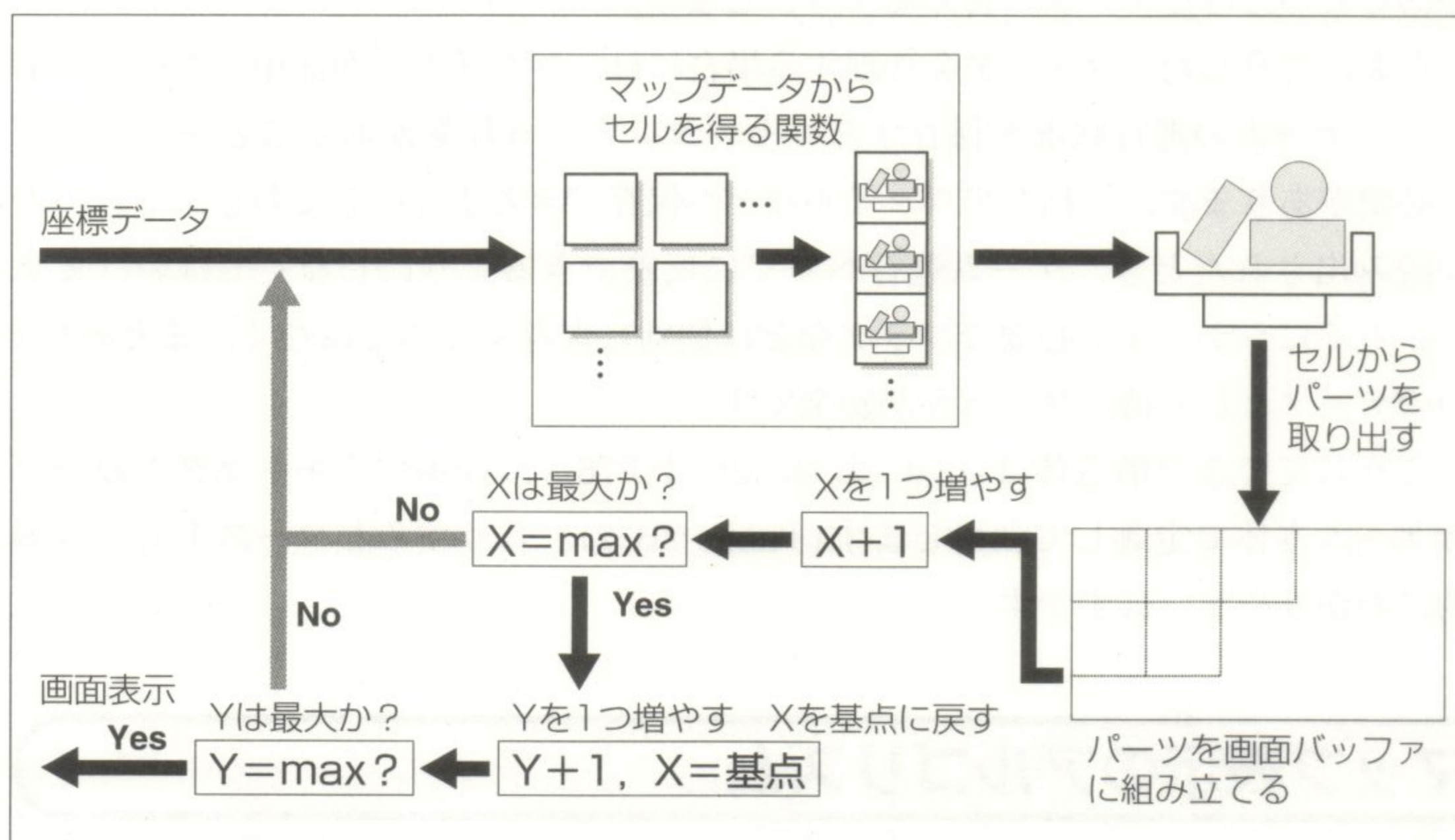



Fig. 2B-7●マップ表示の処理の流れ。マップデータの値からセルを選び出し，セルからパーツを取り出して画面バッファに組み立てる

◆キーに従って通過フラグを調べる

プログラムがユーザからのキー入力を受け取ったら，その方向に移動可能かどうかを判定しければなりません。これは移動したい方向にあるパーツの通過フラグを調べることでわかります。

◆スクロールに従って再描画する

マップのスクロールでは,「全部描き直す」方法と「画像データをスクロールして，必要なパーツだけコピーする」方法があります。スクロール方法はいろいろあ

りますが，スクロールする領域が小さいのならいろいろ手を加えるよりも全部描き直したほうが速くて簡単です。

いずれの方法も「絶対座標」を使っています。マップ左上を0，0として，これからどれだけ離れているかを正の数で指定する方法です。「相対座標」では，現在キャラクタのいる地点などを0，0にして，各方向を指定します。普通は絶対座標で十分ですし直感的にもわかりやすいのですが，あまりにマップが大きく分割してメモリにロードしなくてはいけないようなときは相対座標のほうが便利です。

● マップとキャラクタの合成

キャラクタがほかに何も映っていない画面の中を歩いているだけではゲームになりません。マップとキャラクタを合成して画面上に出力することで，初めて画面にゲームの世界が出現します。合成の方法はChapter1で簡単に説明しましたが，もう一度おさらいしてみましょう。

用意するのは「キャラクタの絵」「背景」「キャラクタのマスクデータ」です。「マスクデータ」は背景を透かして見せたい部分を黒で，そうでないところを白で描きます。次に「背景とマスクデータをAND演算」「それにキャラクタデータをOR演算」という順番でデータを画面にコピーします。この演算はC/C++なら，

```
back &= mask
back |= character
```

Delphiなら，

```
back := back and mask
back := back or character
```

というように書きます。こうしたビット演算(and，or，xor，not)を使ってグラフィックデータを加工することを「ラスタ処理」などと呼んでいます。

◆いろいろな合成方法

マスクデータを使うのは複雑な合成処理をするためです。テレビなどのアニメーションでは透明なプラスチックのフィルム(セル)に絵を描くので，そのまま背景と重ねるだけで合成することができます。一方ピクセルが並んだだけのグラフィックデータでは，いったん背景をキャラクタの輪郭に合わせてくりぬき，そこへキャラ

クタのグラフィックデータをはめ合わせることをしなくてはいけません。マスクデータは，このくりぬき作業のための「型紙」の役目があります。もしマスクデータを作りたくないときは，指定した色があるところを透かすようにする「透過色」や，キャラクタの外周は無条件で背景にする，といった方法もあります。

一部のマシンでは，専用のハードウェアにこのラスタ処理をさせることができます。また「スプライト」と呼ばれる背景専用のグラフィック画面が用意されているマシンもあります。この場合はキャラクタと背景をハードウェア的に分離して描画できるので，あまり合成作業を考えなくて済みます。

Windowsでは，デバイスコンテキスト(DC)へグラフィックデータをコピーするbitblt関数を使って演算処理をしながらコピーすることができます。DelphiやC++BuilderのTCanvasコンポーネントでもこれと同じように「CopyMode」プロパティでコピー方法の指定ができます。最近ではMMXやSIMD，3DNow!を用いたり，CPUパワーが向上したため，コピーしないピクセルを除いて描画する方法のほうが場合によっては高速になります。

キャラクタ表示のアルゴリズム

キャラクタを動かすためのプログラムは，普通は「画面表示」「入力デバイスの統合」「キャラクタ移動」の3つの部分に分けて作りますが，いろいろな機能を持たせる必要があるので，とにかくごちゃごちゃになりがちです。そこで，まずは「動く方向」を引数として受け取ると，それからグラフィックデータを返す処理をする関数だけを考えます。単純に，先ほどの計算式を利用してデータを取り出すだけです。次に，これから得たグラフィックデータをマップ画面と合成する関数を作ります。移動するときには，前に描かれたキャラクタを消すために，元のマップ用パーツデータをコピーする関数も用意します。

こうしたシンプルな機能を持った関数を集めて，キャラクタ移動をするアルゴリズムを考えます(Fig. 2B-8，List 2B-7，2B-8(P81))。始めはキャラクタが描画された地点のマップパーツを元に戻す作業をします。

移動する方向は引数として与えます。この引数のとおりに移動できるかどうかをマップの移動可能フラグを見て判断します。もし移動できるならキャラクタの座標を増減してやります。移動できないのなら，座標は変えずその地点でキャラクタのグラフィックだけを変更します。

引数
移動パターン
移動方向
停止用
パターンか？
移動用だった
X.Y＝Limit？
No
Yes
X.Yが0または
マップの大きさかを
確かめる
停止用
だった
その方向へは
移動可能か？
Yes
No
元の
キャラクタ
を消す
X.Yを
方向に従い増減
X.Y＝
スクロール
エリアか？
No
Yes
マップを
スクロール
元のキャラクタ
を消す
キャラクタと
マップデータを
仮想画面で合成
次の入力を
待つ
キャラクタ分の
マップ用パーツ
をコピー
パターンは
移動用か？
Yes
パターン
切り換え
No
グラフィック
データを合成
（ビット演算などを利用）
キャラクタ用
グラフィックデータ
を取得

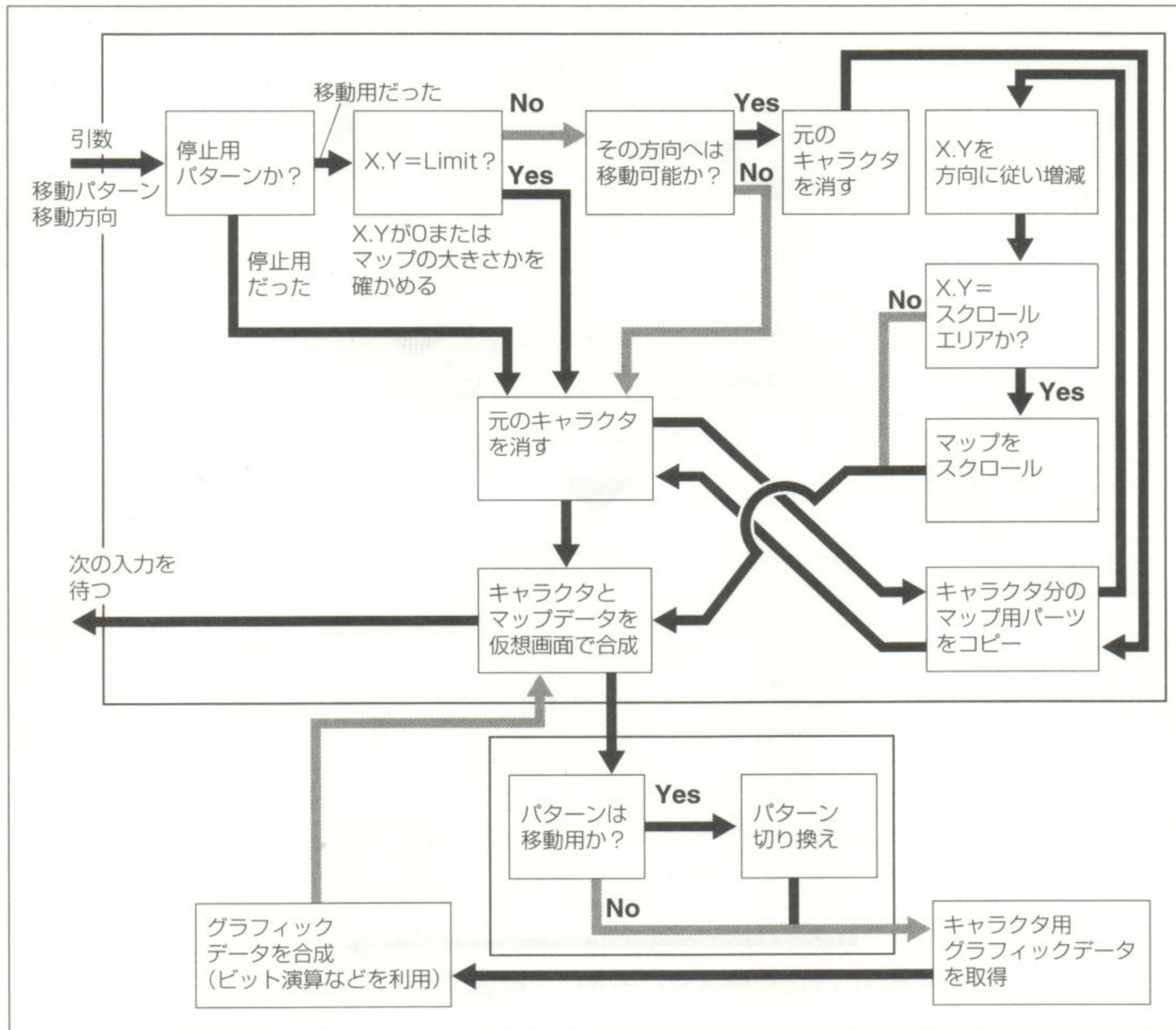



Fig. 2B-8 ●キャラクタ表示のアルゴリズム

◆移動に合わせてマップをスクロールする

マップ画面の端にキャラクタがきた場合，マップをスクロールさせなくてはなりません。ぎりぎり端にきたときにスクロールさせるのではなく，1パーツ以上の間を空けてスクロールさせます(Fig. 2B-9)。

この処理をしたあとに新しくキャラクタを描きます。もし移動に使うパターンを取り出したいのなら，前述の変数を参照して取り出すグラフィックを決めます。あとは与えられた引数とキャラクタ用データを基にマップ画面とキャラクタパターンを合成します。

キャラクタ表示部分では，このほかにも移動先にイベントがあったらそれを処理する関数など，キャラクタと連動させたいものはこの関数の中に加えていきます。

キャラクタが画面端にくると

キャラクタの位置は変わらずに，背景がスクロールする

Fig. 2B-9●キャラクタが画面の端手前の位置にきたら，キャラクタの位置はそのままで，背景を進行方向と逆にスクロールさせる

◆入力用デバイスの処理は統一する

最後に「入力の統一」部分です。ゲームで方向を指示する機器には「マウス」「キーボード」「ジョイスティック」があり，これらの入力処理はほとんどばらばらです。入力された方向が「上」なら上としてまとめ，これをキャラクタ移動用関数に渡してやります。

● サンプルゲームの遊び方

サンプルゲームは普通のRPGとはちょっと違っていて，敵ボスはプレイヤーの動かすキャラクタと同じようにマップ上を移動しています。これをオニゴッコのように捕まえて倒すのがゲームの目的です。世界観は「登場する人間はみんなバニ

ーさん」ということにしてコミカルなものにしました。いろいろなイベントを用意したので，ぜひ楽しんでみてください。

パーツデータは32×32ドットを基本の単位にしています。マップデータやキャラクタデータはファイルとして収納しています。これらを起動時に確保したメモリへまとめて読み込んでいます。

フラグやイベントは気をつけないとプログラミング中何回修正を繰り返しても，思ったようにうまく動かないという泥沼の状態になります。できるだけこれらをシナリオデータなどに記述して，シナリオの相関状態はプログラム側には含めないようにすると，この泥沼を回避することができるでしょう。

＊

RPGは，ほかのゲームと比べても長時間熱中できるゲームです。RPGのために徹夜したという経験がある人は多いかもしれません。これはまるでジグソーパズルのようです。もしRPGが好きなら，ぜひ自分でRPGを作ってみてください。さらにおもしろい世界が広がると思います。

List 2B-1 ●マップのセル（Delphi）

```
{ 型宣言 }
type
   cell = record                { 枠(セル)の定義     }
      gra : ^byte;              { グラフィックデータ }
      event_flag : integer;     { イベントフラグ     }
      move_flag : integer;      { 移動可能フラグ     }
   end;
```

List 2B-2 ●マップのセル（C/C++）

```
/* 型宣言 */
typedef struct {              /* 枠(セル)の定義     */
   unsigned char *gra;        /* グラフィックデータ */
   int event_flag;            /* イベントフラグ     */
   int move_flag;             /* 移動可能フラグ     */
} cell_t;
```

List 2B-3●キャラクタデータ用の構造体(Delphi)

```
{ 定数宣言 }
const
  CHAR_STOP = 0;
  CHAR_STOPMASK = 1;
  CHAR_MOVE = 2;
  CHAR_MOVEMASK = 3;
  CHAR_MOVE1 = 4;
  CHAR_MOVE1MASK = 5;
  CHAR_MOVEMAX = 6;

  CHAR_FACE =  0;
  CHAR_BATL1 = 1;
  CHAR_BATL2 = 2;
  CHAR_BATL3 = 3;
  CHAR_MAX =   4;

  CHAR_UP = 0;
  CHAR_DOWN = 1;
  CHAR_LEFT = 2;
  CHAR_RIGHT = 3;
  CHAR_OTHER = 4;

  CHAR_TBLMAX = 28;

{ 型宣言 }
type
    TCharacter = record                         { 構造体の定義        }
        gra : array[0..CHAR_TBLMAX] of ^byte;   { グラフィックデータ  }
        Name: String;                           { キャラクタ名        }
        HP: integer;                            { 体力                }
        HPmax: integer;                         { 体力最大            }
        MP: integer;                            { 魔力                }
        MPmax: integer;                         { 魔力最大            }
        x: integer;                             { マップ上の横位置    }
        y: integer;                             { マップ上の縦位置    }
        ViewX: integer;                         { 画面上の横位置      }
        ViewY: integer;                         { 画面上の縦位置      }
        SelectPattern: integer;                 { 前のパターンデータ  }
    end;
```

List 2B-4●キャラクタデータ用の構造体(C/C++)

```
/* 定数宣言 */
#define CHAR_STOP       0
#define CHAR_STOPMASK   1
#define CHAR_MOVE       2
```

```
#define CHAR_MOVEMASK   3
#define CHAR_MOVE1      4
#define CHAR_MOVE1MASK  5
#define CHAR_MOVEMAX    6

#define CHAR_FACE       0
#define CHAR_BATL1      1
#define CHAR_BATL2      2
#define CHAR_BATL3      3
#define CHAR_MAX        4

#define CHAR_UP         0
#define CHAR_DOWN       1
#define CHAR_LEFT       2
#define CHAR_RIGHT      3
#define CHAR_OTHER      4

#define CHAR_TBLMAX     28

/* 型宣言 */
typedef struct {                        /* 枠(セル)の定義      */
    char gra[CHAR_TBLMAX + 1];          /* グラフィックデータ */
    char Name[255];                     /* キャラクタ名       */
    int HP;                             /* 体力               */
    int HPmax;                          /* 体力最大           */
    int MP;                             /* 魔力               */
    int MPmax;                          /* 魔力最大           */
    int x;                              /* マップ上の横位置   */
    int y;                              /* マップ上の縦位置   */
    int ViewX;                          /* 画面上の横位置     */
    int ViewY;                          /* 画面上の縦位置     */
    int SelectPattern;                  /* 前のパターンデータ */
} character_t;
```

List 2B-5 ●マップデータ表示(Delphi)

```
{ 型宣言 }
type
   point_t = record          { 座標指定用構造体         }
      x : integer;
      y : integer;
   end;
   point_ptr = ^point_t;
   scroll_t = record          { スクロール指定用構造体   }
      x : integer;
      y : integer;
   end;
   scroll_ptr = ^scroll_t;
```

List 2B-5

```
    cell_ptr = ^cell_t;

{ マップデータ }
var
    celltbl : array[0..16] of cell_t;         { セルテーブル }
    mapdata : array[0..64,0..64] of byte;  { マップデータ }
    partssize : point_t;                       { パーツサイズ }

{ マップデータから特定のセルを取得 }
function rpg_map_getcell(mapofs : point_ptr) : cell_ptr;
var
    ofs : byte;
begin
    ofs := mapdata[mapofs^.x][mapofs^.y];  { セルテーブルの位置を取得 }
    rpg_map_getcell := @celltbl[ofs];       { セルデータを取り出す     }
end;

{ マップデータの表示 }
procedure rpg_map_view(mapofs : point_ptr; mapviewsize : point_ptr);
var
    tmpofs : point_t;
    cellp : cell_ptr;
    x, y : integer;
begin
    tmpofs.y := mapofs^.y;
    tmpofs.x := mapofs^.x;
    for y := 0 to mapviewsize^.y do begin
        for x := 0 to mapviewsize^.x do begin
            cellp := rpg_map_getcell(@tmpofs);  { セルを取得 }
            vbuf_copy(@tmpofs, cellp, @partssize);  { 画面バッファへコピー }
            inc(tmpofs.x);
        end;
        tmpofs.x := mapofs^.x;  { xを基点に戻す }
        inc(tmpofs.y);
    end;
end;

{ 通行可能かどうか調べる }
function rpg_map_ismove(mapofs : point_ptr) : integer;
var
    cellp : cell_ptr;
begin
    cellp := rpg_map_getcell(mapofs);     { 指定された地点のセルを取得 }
    rpg_map_ismove := cellp^.move_flag;  { 通過フラグを返す             }
end;

{ マップデータのスクロール }
procedure rpg_map_scr(mapofs : point_ptr; mapviewsize : point_ptr;
                         mapsize : point_ptr; scroll : scroll_ptr);
var
```

```
    tmpofs : integer;
begin
    if scroll^.y <> 0 then begin
{ 縦方向計算    }
        tmpofs := mapofs^.y + scroll^.y;
        if tmpofs < 0 then
                { 範囲チェック      }
            tmpofs := 0
        else if tmpofs >= mapsize^.y then
            tmpofs := mapsize^.y - 1;
        mapofs^.y := tmpofs;
    end;
    if scroll^.x <> 0 then begin
{ 横方向計算       }
        tmpofs := mapofs^.x + scroll^.x;
        if tmpofs < 0 then
                { 範囲チェック      }
            tmpofs := 0
        else if tmpofs >= mapsize^.x then
            tmpofs := mapsize^.x - 1;
        mapofs^.x := tmpofs;
    end;
    rpg_map_view(mapofs, mapviewsize);  { パーツコピー }
end;
```

List 2B-6 ●マップデータ表示(C/C++)

```
/* 座標指定用構造体 */
typedef struct {
    int x;
    int y;
} point_t;

/* スクロール指定用構造体 */
typedef struct {
    int x;          /* 横移動サイズ(正数で右へ、負数で左へ) */
    int y;          /* 縦移動サイズ(正数で下へ、負数で上へ) */
} scroll_t;

/* マップデータ */
static cell_t celltbl[16 + 1];                      /* セルテーブル */
static unsigned char mapdata[64 + 1][64 + 1];       /* マップデータ */
static point_t partssize;                           /* パーツサイズ */

/* マップデータから特定のセルを取得    */
/*                                      */
/* mapofs:    マップの位置              */
cell_t *rpg_map_getcell(point_t *mapofs)
{
```

List 2B-6

```
    unsigned char ofs;

    ofs = mapdata[mapofs->x][mapofs->y];  /* セルテーブルの位置を取得  */
    return &celltbl[ofs];                  /* セルデータを取り出す      */
}

/* マップデータの表示                     */
/*                                         */
/* mapofs:     マップの位置                */
/* mapviewsize:表示する大きさ              */
void rpg_map_view(point_t *mapofs, point_t *mapviewsize)
{
    point_t tmpofs;
    cell_t *cellp;
    int x, y;

    tmpofs.y = mapofs->y;
    tmpofs.x = mapofs->x;
    for (y = 0; y < mapviewsize->y; y++, tmpofs.y++) {
        for (x = 0; x < mapviewsize->x; x++, tmpofs.x++) {
            cellp = rpg_map_getcell(&tmpofs);
                                              /* セルを取得          */
            vbuf_copy(&tmpofs, cellp, &partssize);
                                              /* 画面バッファへコピー */
        }
    tmpofs.x = mapofs->x;          /* x を基点に戻す    */
        }
}

/* その地点が通行可能かどうか調べる */
/*                                  */
/* mapofs:     マップの位置         */
int rpg_map_ismove(point_t *mapofs)
{
    cell_t *cellp;

    cellp = rpg_map_getcell(mapofs);  /* 指定された地点のセルを取得  */
    return cellp->move_flag;          /* 通過フラグを返す            */
}

/* マップデータのスクロール         */
/* (全画面書き換え)                 */
/*                                  */
/* mapofs:        マップの位置      */
/* mapviewsize:表示する大きさ       */
/* mapsize:       マップの大きさ    */
/* scroll:        スクロールデータ  */
void rpg_map_scr(point_t *mapofs, point_t *mapviewsize,
                                   point_t *mapsize, scroll_t *scroll)
{
```

```
    int tmpofs;

    if (scroll->y) {                  /* 縦方向計算    */
        tmpofs = mapofs->y + scroll->y;
        if (tmpofs < 0) {             /* 範囲チェック  */
            tmpofs = 0;
        } else if (tmpofs >= mapsize->y) {
            tmpofs = mapsize->y - 1;
        }
        mapofs->y = tmpofs;
    }
    if (scroll->x) {                  /* 横方向計算    */
        tmpofs = mapofs->x + scroll->x;
        if (tmpofs < 0) {             /* 範囲チェック  */
            tmpofs = 0;
        } else if (tmpofs >= mapsize->x) {
            tmpofs = mapsize->x - 1;
        }
        mapofs->x = tmpofs;
    }
    rpg_map_view(mapofs, mapviewsize);  /* パーツコピー  */
}
```

List 2B-7●キャラクタデータの表示(Delphi)

```
(* ------------------------------------------------ *)
const
  CHARCMD_UP = 0;
  CHARCMD_DOWN = 1;
  CHARCMD_LEFT = 2;
  CHARCMD_RIGHT = 3;
  CHARCMD_STOP = 4;

  VIEWSCRAREA = 1;


var
    Okano: TCharacter;                        { キャラクタデータ       }
    OldDirection: integer;                    { 前の移動方向           }
    MapViewX, MapViewY: integer;              { 表示するマップの位置   }
    MapWidth, MapHeight: integer;             { マップの大きさ         }
    MapViewWidth, MapViewHeight: integer;     { マップ画面の大きさ     }


(* ------------------------------------------------ *)
{ スクロールの判定をしながらキャラクタ描画 }
procedure ScrollToMap(Direction, Pattern,
 ScrollCmd, ScrollArea, DifX, DifY: Integer);
```

List 2B-7

```
var
    NewX, NewY: integer;
begin
    MapDraw(Okano.x, Okano.y, MapViewX, MapViewY); { 前地点のキャラを消す }
    NewX := Okano.x + DifX;
    NewY := Okano.y + DifY;
    if MapGetMoveflag(NewX, NewY) then begin          { そこへ移動できるか？ }
        if DifX <> 0 then begin
            Okano.x := NewX;
            if (Okano.ViewX = ScrollArea) and         { スクロール判定        }
                (Okano.x <> 0) and (Okano.x <> MapWidth - 1) then begin
                MapScroll(ScrollCmd);                 { マップスクロール      }
            end else begin
                Okano.ViewX := Okano.ViewX + DifX;
            end;
        end else if DifY <> 0 then begin
            Okano.y := NewY;
            if (Okano.ViewY = ScrollArea) and
               (Okano.y <> 0) and (Okano.y <> MapHeight - 1) then begin

                MapScroll(ScrollCmd);
            end else begin
                Okano.ViewY := Okano.ViewY + Dify;
            end;
        end;
    end;
    RopDraw(Direction, Pattern);     { マップ画面とキャラクタを合成 }
end;

{ 指定されたマップの地点にキャラクタを描画 }
procedure DrawToMap(Direction, Pattern: Integer);
begin
    if MapGetMoveflag(Okano.x, Okano.y) then begin
        MapDraw(Okano.x, Okano.y, MapViewX, MapViewY);
                                                { 前地点のキャラを消す }
        RopDraw(Direction, Pattern);    { マップ画面とキャラクタを合成 }
    end;
end;

{ マップ画面へキャラクタを出力してマップのスクロールもする }
procedure CharacterScroll(Cmd: integer);
begin
    case Cmd of
      CHARCMD_UP : begin
        if Okano.y > 0 then begin
        ScrollToMap(CHAR_UP, CHAR_MOVE, SCROLL_UP, VIEWSCRAREA, 0, -1);

        end else begin
            DrawToMap(CHAR_UP, CHAR_MOVE);
        end;
```

```
      end;
      CHARCMD_DOWN : begin
        if Okano.y < MapHeight then begin
            ScrollToMap(CHAR_DOWN, CHAR_MOVE, SCROLL_DOWN,
              MapViewHeight - 1 - VIEWSCRAREA, 0, 1);
        end else begin
            DrawToMap(CHAR_DOWN, CHAR_MOVE);
        end;
      end;
      CHARCMD_LEFT : begin
        if Okano.x > 0 then begin
    ScrollToMap(CHAR_LEFT, CHAR_MOVE, SCROLL_LEFT, VIEWSCRAREA, -1, 0);

        end else begin
            DrawToMap(CHAR_LEFT, CHAR_MOVE);
        end;
      end;
      CHARCMD_RIGHT : begin
        if Okano.x < MapWidth then begin
            ScrollToMap(CHAR_RIGHT, CHAR_MOVE, SCROLL_RIGHT,
              MapViewWidth - 1 - VIEWSCRAREA, 1, 0);
        end else begin
            DrawToMap(CHAR_RIGHT, CHAR_MOVE);
        end;
      end;
      CHARCMD_STOP : begin
        DrawToMap(OldDirection, CHAR_STOP);
      end;
    end;
end;

end.
```

List 2B-8●キャラクタデータの表示(C/C++)

```
/* ----------------------------------------------- */
#define CHARCMD_UP      0
#define CHARCMD_DOWN    1
#define CHARCMD_LEFT    2
#define CHARCMD_RIGHT   3
#define CHARCMD_STOP    4

#define VIEWSCRAREA     1

static character_t Okano;                  /* キャラクタデータ      */
static int OldDirection;                   /* 前の移動方向          */
static int MapViewX, MapViewY;             /* 表示するマップの位置  */
static int MapWidth, MapHeight;            /* マップの大きさ        */
static int MapViewWidth, MapViewHeight;    /* マップ画面の大きさ    */
```

List 2B-8

```
/* ---------------------------------------------- */
/* スクロールの判定をしながらキャラクタ描画 */
void ScrollToMap(int Direction, int Pattern,
 int ScrollCmd, int ScrollArea, int DifX, int DifY)
{
    int NewX, NewY;

    MapDraw(Okano.x, Okano.y, MapViewX, MapViewY);   /* 前地点のキャラを消す */
     NewX = Okano.x + DifX;
     NewY = Okano.y + DifY;
    if (MapGetMoveflag(NewX, NewY)) {                /* そこへ移動できるか? */
         if (DifX != 0) {
             Okano.x = NewX;
             if ((Okano.ViewX == ScrollArea) &&       /* スクロール判定 */
               (Okano.x != 0) && (Okano.x != (MapWidth - 1))) {
               MapScroll(ScrollCmd);                  /* マップスクロール */
             } else {
                 Okano.ViewX = Okano.ViewX + DifX;
             }
         } else if (DifY != 0) {
             Okano.y = NewY;
             if ((Okano.ViewY = ScrollArea) &&
               (Okano.y != 0) && (Okano.y != (MapHeight - 1))) {
                 MapScroll(ScrollCmd);
             } else {
                 Okano.ViewY = Okano.ViewY + DifY;
             }
         }
    }
     RopDraw(Direction, Pattern);                     /* マップ画面とキャラクタを合成 */
}

/* 指定されたマップの地点にキャラクタを描画 */
void DrawToMap(int Direction, int Pattern)
{
     if (MapGetMoveflag(Okano.x, Okano.y)) {
        MapDraw(Okano.x, Okano.y, MapViewX, MapViewY); /*前地点のキャラを消す*/
        RopDraw(Direction, Pattern);              /* マップ画面とキャラクタを合成*/
     }
}

/* マップ画面へキャラクタを出力してマップのスクロールもする */
void CharacterScroll(int Cmd)
{
     switch(Cmd) {
       case CHARCMD_UP:
        if (Okano.y > 0) {                        /* マップの大きさを判断して描画 */
         ScrollToMap(CHAR_UP, CHAR_MOVE, SCROLL_UP, VIEWSCRAREA, 0, -1);
         } else {
```

```
                DrawToMap(CHAR_UP, CHAR_MOVE);
            }
            break;
        case CHARCMD_DOWN:
            if (Okano.y < MapHeight) {
                ScrollToMap(CHAR_DOWN, CHAR_MOVE, SCROLL_DOWN,
                  MapViewHeight - 1 - VIEWSCRAREA, 0, 1);
            } else {
                DrawToMap(CHAR_DOWN, CHAR_MOVE);
            }
            break;
        case CHARCMD_LEFT:
            if (Okano.x > 0) {
               ScrollToMap(CHAR_LEFT, CHAR_MOVE, SCROLL_LEFT, VIEWSCRAREA, -1, 0);
            } else {
                DrawToMap(CHAR_LEFT, CHAR_MOVE);
            }
            break;
        case CHARCMD_RIGHT:
            if (Okano.x < MapWidth) {
                ScrollToMap(CHAR_RIGHT, CHAR_MOVE, SCROLL_RIGHT,
                  MapViewWidth - 1 - VIEWSCRAREA, 1, 0);
            } else {
                DrawToMap(CHAR_RIGHT, CHAR_MOVE);
            }
            break;
        case CHARCMD_STOP:
            DrawToMap(OldDirection, CHAR_STOP);
            break;
    }
}
```

Section 3 ダンジョンゲーム

ダンジョンゲームは擬似的に作られた迷路の中を歩くことができるおもしろいゲームです。このダンジョンゲームで使われるアルゴリズムとデータ構造について，とくに迷路の自動生成と表示方法について考えます。

● 迷うことを楽しもう

人によって方向音痴の程度はさまざまのようです。私の場合はお店などの位置を「空間として把握」しているのですが，お店に至るまでの「道順」はぜんぜん覚えていません。その結果どうなるかというと，地図がなければ「店の周囲をぐるぐる回る」という情けないことになります。最終的にはちゃんと目的の場所へたどり着きますが，これに知り合いが同行するとただの散歩になってしまい，なんとももうしわけないかぎりです。

こんな「よくぐるぐる迷う」私がスキーへいくともうたいへんです。初めていったゲレンデだと，とにかくコースを覚えません。目的のレストハウスに着かないなんてことはよくあります。とりあえず誰かに先にいってもらって先導するようにいつもお願いしています。そうひどい方向音痴とは思っていないのですが，私にとってはああいうものこそ迷路です。

このように「現在いる場所がわからない」という焦燥感をゲームにしたのが「ダンジョンゲーム」です。ここでは，このゲームを題材にして迷路の自動生成や表示などのアルゴリズムを取りあげます。

● ダンジョンゲームとは？

迷宮の中をプレイヤーが歩き回り，途中でアイテムを拾ったり，モンスターを倒しながら目標へと進むのがダンジョンゲームです。このゲームの最大のポイントは「迷路の中を歩く」ことにあります。入り組んだ通路の中を出口を求めてさまよう，迷路独特の「不安」と「焦燥」や，出口にたどり着いたときの「達成感」をゲームとして体験することができます。

◆ダンジョンゲームのタイプ

ダンジョンゲームは，ゲーム画面の「視点」で2つのタイプに分けることができます。まずは「迷路を上から見る視点」のものです。このタイプでは「Rouge」や「NetHack」などキャラクタベースで作られたゲームが有名で，とても根強い人気があります。一方で「ゲームの中にいるキャラクタの視点」で迷路を表示するものもあります。現在ダンジョンゲームといえば，このタイプのゲームが大半を占めています。表示される迷路の画像は，初期のころはただのワイヤフレームでしたが，最近ではCPUやグラフィック処理性能の向上とともに，壁などの絵が表示されるようなり，臨場感あふれる画面になりました。さらに現在では「DOOM」などに見られるスムーズに動く自然な方法が使われています。

ストーリには「剣と魔法の世界」といったファンタジー系のものが多く見られます。現実的なところでは，たとえば学校などを迷路として考えることができるでしょう。実際に，そうした題材を選んでいるゲームもあります。最近では「ダンジョン」だけのゲームよりも，アドベンチャーゲームやロールプレイングゲームの要素と複合して作られているゲームのほうが主流です。

◆迷路ならではのおもしろさ

このように，「作られたゲームの世界を歩く」「敵を倒す」「目的を果たす」という面では，ロールプレイングゲームやアドベンチャーゲームなど，ほかのゲームと多くの共通点があります。ではダンジョンゲームでなければ味わうことのできないおもしろさはあるのでしょうか？　ダンジョンゲームでは視点を「迷路の中」に求めることで，より現実に近い「いまどこを歩いているのかわからなくなる」という感覚を出すことに成功しています。ここにダンジョンゲーム本来のおもしろさがあります。この感覚があるからこそ，目的を達成したときの感動がより濃くなるのです。

こうした「迷路を解くことそのものを楽しむ」ことにより「ほかのゲーム性がより際立つ」のがダンジョンゲームの特徴です。

● ダンジョンゲームの中身

「キャラクタの視点」で表示されるタイプのゲームを例にして，実際にプレイしながらダンジョンゲームの仕組みを考えてみます。

画面にはキャラクタの見ている迷路のようすが描かれます(Fig. 2C-1)。暗くじ

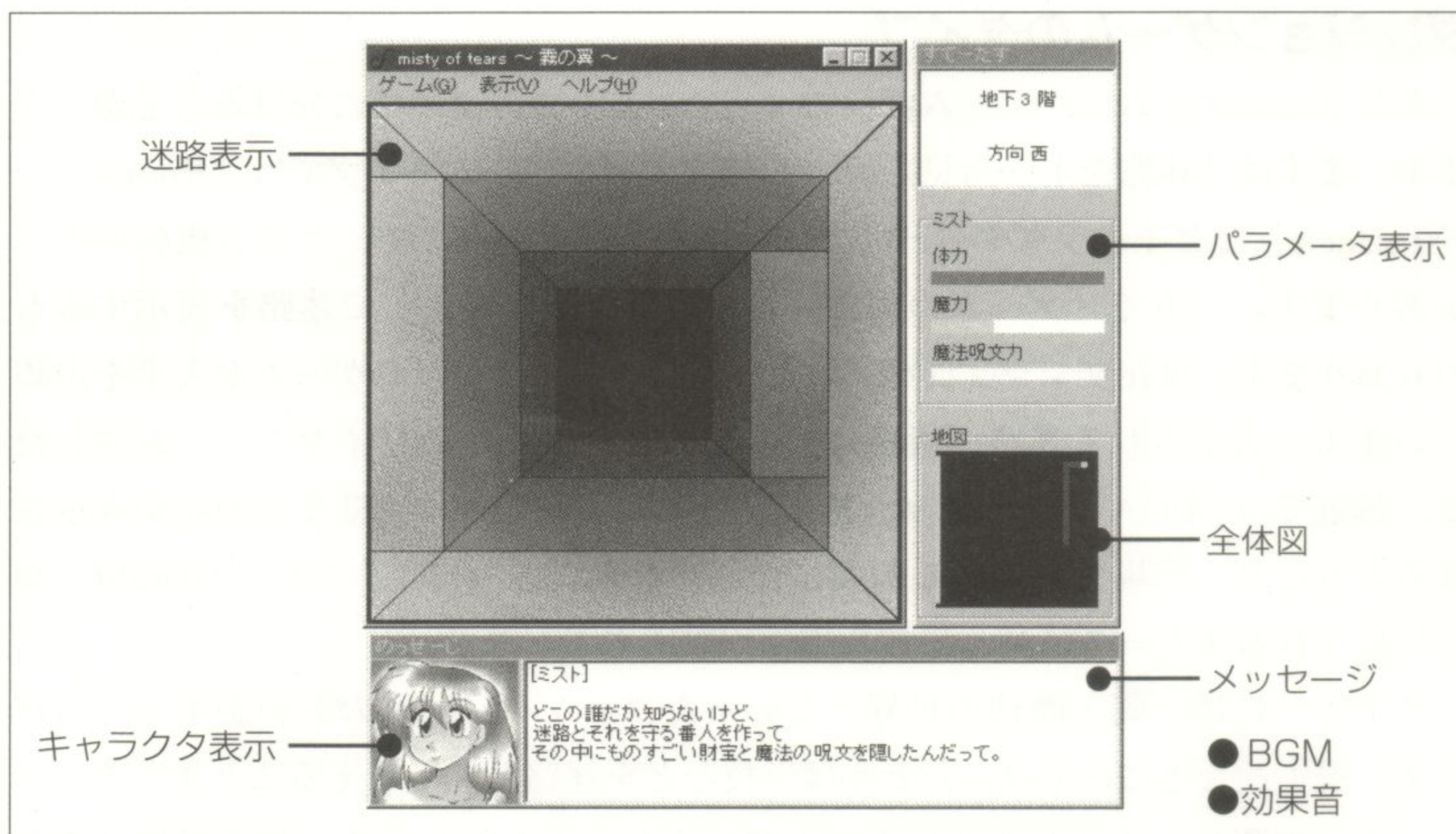



Fig. 2C-1 ●ダンジョンゲームの画面。ゲームプレイ時にはBGMと効果音が加わる

めじめした迷路の中の陰鬱な風景がプレイヤーの前に広がっています。よく見ると，壁は遠近法に従って奥が狭く，手前が広く見えているようです。

迷路そのものはデータとしてゲームに格納されているはずです。壁の表示は，この迷路データを調べることから始まります。つまり，表示画面を作るには「キャラクタが向いている方向に存在している壁の有無」を調べることが必要です。画面表示のアルゴリズムはこの壁を調べることが中心になります。

◆「位置」と「方向」が重要

プレイヤーがキーを押して操作すると，それに合わせて画面も変化します。キャラクタが前に移動すれば，一歩先にある壁が手前へと迫ってくるように表示されます。

まずは，迷路の中にいるキャラクタの「位置」というものを考えてみます。現在キャラクタがいる位置は「座標」という値で示すことができます。キャラクタが前へ動くということは，位置と対応する座標の値も変化するはずです。ただし，このままでは「キャラクタが前へ動いた」ことはわかっても，「移動先の座標」はわかりません。移動先の座標を知るためには，キャラクタが向いている「方向」を考えなければいけないからです。

キャラクタが北を向いているときに前へと踏み出したのなら，移動先のキャラクタの位置は，元の位置よりも北寄りになっています。これを座標の値の増減とし

て考えると，マップの左上が「0，0」になっているとして，「縦＝縦-1」というように，座標の値が減ることになります(Fig. 2C-2)。南ならその逆に値を増やし，東西なら横の座標が増減します。こうした「キャラクタの位置から見た座標」を「相対座標」と呼んでいます。

一方，マップデータ上で「原点となる座標(この場合は左上隅)から見た座標」を「絶対座標」といいます。相対座標から絶対座標への変換は，「向いている方向」「動く方向」の2つの値と元の位置との「マップデータ上の座標の差」を取り出す専用のテーブルを用意します。先ほどの「向いているのが北」「動くのが前」なら，このテーブルから引き出せる値は「X＝0，Y＝-1」という形になります(迷路の横軸をX，縦軸をYとする)。あとはこれにキャラクタが元いたマップ上の座標を加えてやれば，その位置にある迷路のデータをすぐに取り出すことができます。

0, 0
北
前
東
左
右
西
後
南
5, 5
X=3，Y=3
向いている方向=北

	相対座標	絶対座標
前	X=0，　Y=-1	X=3，Y=2
後	X=0，　Y=1	X=3，Y=4
左	X=-1，　Y=0	X=2，Y=3
右	X=1，　Y=0	X=4，Y=3

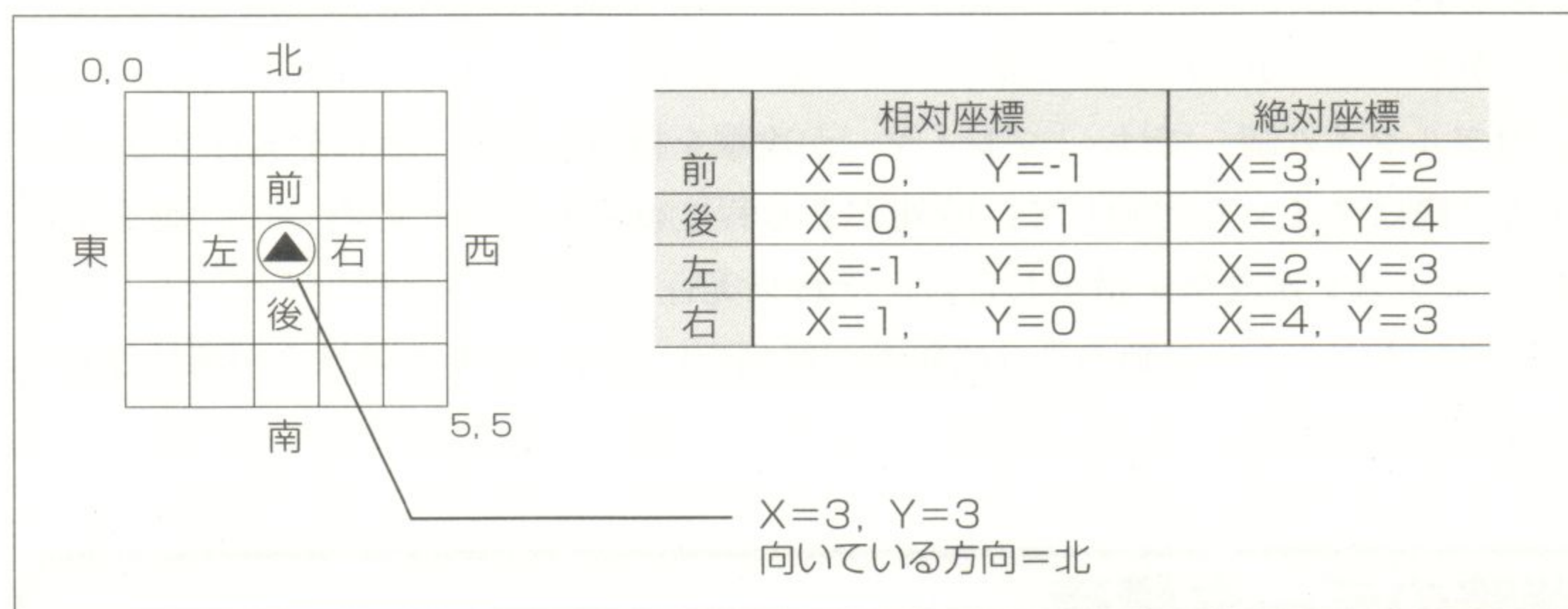



Fig. 2C-2●迷路の左上隅を原点(0，0)として，キャラクタが現在いる位置の座標と向いている方向を調べる。その位置からの移動先を相対座標として取得し，それを絶対座標に変換する

◆迷路の中でイベントを発生させる

迷路の中をしばらく歩くと，モンスタに出会ったり，アイテムや階段などが見つかることもあります。これらはイベントとして処理されます。キャラクタの座標が変わったときに，その移動先の地点でイベントがあるかどうかを判断して，もしイベントがあるようなら対応するイベント用の処理ルーチンを呼び出します。リアルタイムでモンスタが動くタイプでなければ，この時点でモンスタの移動処理なども実行しています。

◆戦闘のシステム

モンスタに出会うと戦闘が始まります。体力などがなくなればゲームオーバになるのですが，ここで使われる「体力」「魔力」といったものは各キャラクタごとに「パラメータ」と呼ばれる変数にされています。戦闘とは，このパラメータを操作して，ある条件になったら勝敗を決めているだけにすぎません。また取得したアイテムなども，このパラメータの1つとして管理されます。

戦闘のシステムは「ターン制」なら時間や操作する順番によって，「関数ポインタ」と「カウンタ」を利用して処理を切り換えていきます。たとえば最初の「モンスタが出た」などの表示は「カウンタ＝0」として，そのカウンタの値に対応する処理を関数ポインタから実行します。同様に「プレイヤーの攻撃選択」を「カウンタ＝1」と対応させていき，最後のモンスタの攻撃が終わったら，再び「カウンタ＝1」としてプレイヤーの攻撃選択のターンを実行します。これをゲームオーバーかモンスタが倒れるまで繰り返して進めます。

「リアルタイム制」では，「プレイヤーが攻撃を指示したら，それを実行する」「モンスタ側はタイマを起動して，時間がある程度経過したらそのたびに行動を決める」「モンスタが攻撃を指示したら，それを実行する」といった「イベント」の形で処理します。ターン制のように順番が関係ないので，こちらのほうが楽に作れるかもしれません。

迷路のデータ構造

ダンジョンゲーム特有の処理としては「画面表示」と「迷路データの管理」があります。まずこの処理について調べてみましょう。

迷路のデータ構造は，ロールプレイングゲームのマップデータと同じにすることができます。「どこに壁があるのか」「道はどう続いているのか」を表すために2次元配列を使います(Fig. 2C-3，List 2C-1，2C-2(P99))。

「通路」を作るときは，対応する値をこの配列へ入れます。たとえば通路を示す値が「0」で，「横5マス」「縦10マス」のところに通路を作りたいのなら，Delphiでは，

```
mazedata[5][10] := 0;
```

C/C++では，

```
mazedata[5][10] = 0;
```

というように設定します。「壁」を置きたいのなら，同じように壁を示す値(たとえば「1」)を置きます。

これだけでも迷路のデータとして十分使えますが，実際のゲームでは扉や階段といったイベントを配置しなくてはいけません。そこで「1＝壁」のように「値が直接意味になる」のではなく，「1＝テーブルの引数」として，実際に壁であるかどうかは，このテーブルを参照することでわかるようにします。こうしたクッショ

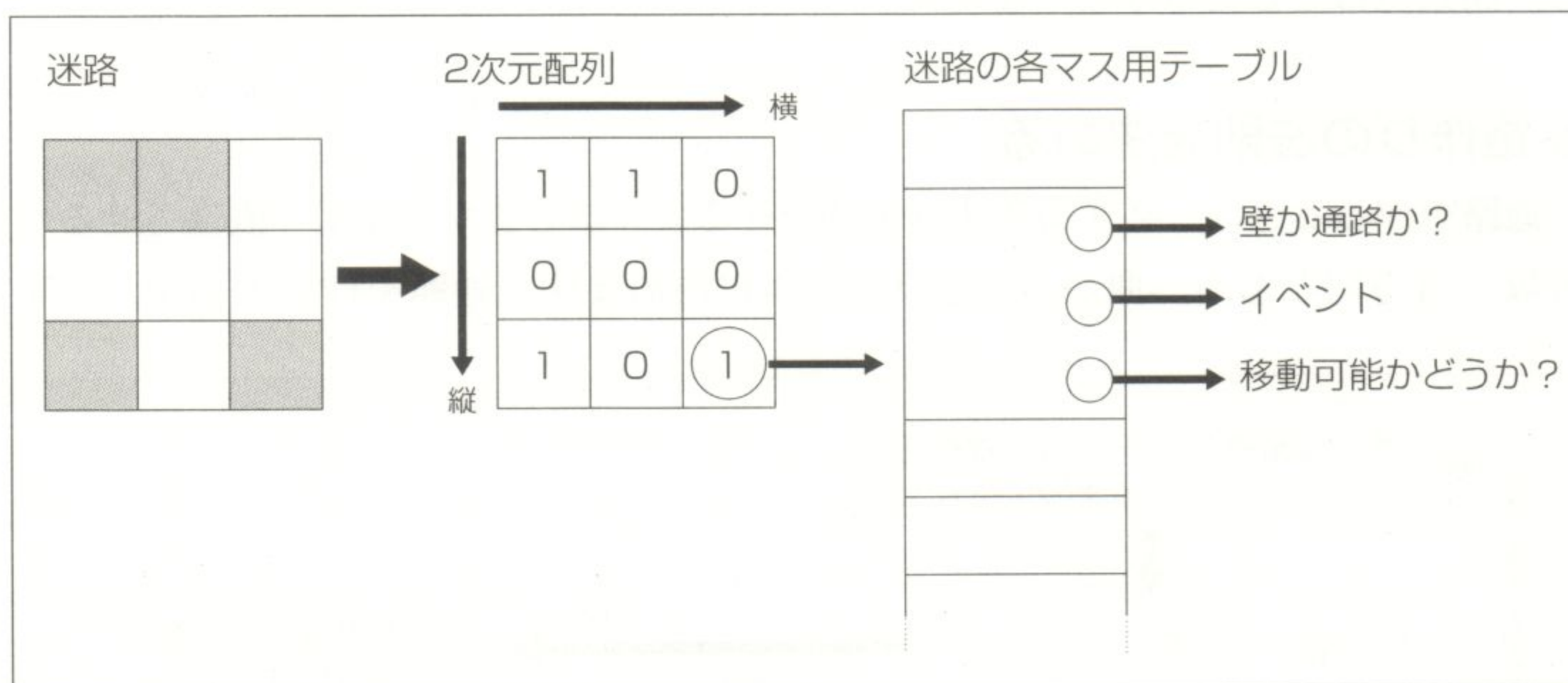



Fig. 2C-3●迷路を２次元配列に置き換え，各マスごとに用意されているテーブルからイベントなどのさまざまなデータを参照する

ンを入れることで，より多くのイベントやデータを管理することができます。また，テーブルの引数としなくても，マップデータの値だけでイベントを表すようにしてもよいでしょう。

◆キャラクタデータの管理

キャラクタのデータも，ロールプレイングゲームと似たようなものです。体力や戦闘力などを，それぞれ変数としてキャラクタパラメータ用の構造体に定義します。量が少なければ，ばらばらに管理してもかまいません。List 2C-3，2C-4(P100)は魔法処理用に定義したパラメータの構造体です。魔法の言葉とこの構造体に必要なものを定義して魔法処理用の関数に渡せば，その結果が構造体に含まれて返ってくる仕組みを想定しています。

● 迷路生成のアルゴリズム

迷路を自動的に生成する場合は，通路を置いていく際に「壁にぶつかったらランダムに方向を決めて曲がる」「ある程度直線が続いたらランダムに曲がる」の2つが基本的なアルゴリズムになります(Fig. 2C-4)。今回は整然とした迷路ではなく洞窟のように乱れた形にしてみます。

まず始めに縦横が決まったサイズの2次元配列を用意します。次に「開始点(通路を作り始める位置)」をランダムに選びます。入口と出口を決めなくてよいのなら，開始点はどこでもかまいません。

◆迷路作りの法則を決める

通路を作るときは，必ずある法則を持たせるようにします。まず「直進させるマス数」「4方向どちらへ曲がる」をランダムに決めます。通路を作る方向が，いま

適当な座標をランダムに決める
部屋があるなら部屋の位置から
直進させる数になったか？
Yes
方向と直進させる数を
ランダムに決める
No
2マス先は通路か？
Yes
のような場合
No
通路を作る
Mazedata[X][Y]に通路
を表すデータを代入
座標を方向に従って増減
No
通路を指定された数だけ
作ったか？
Yes
完　成

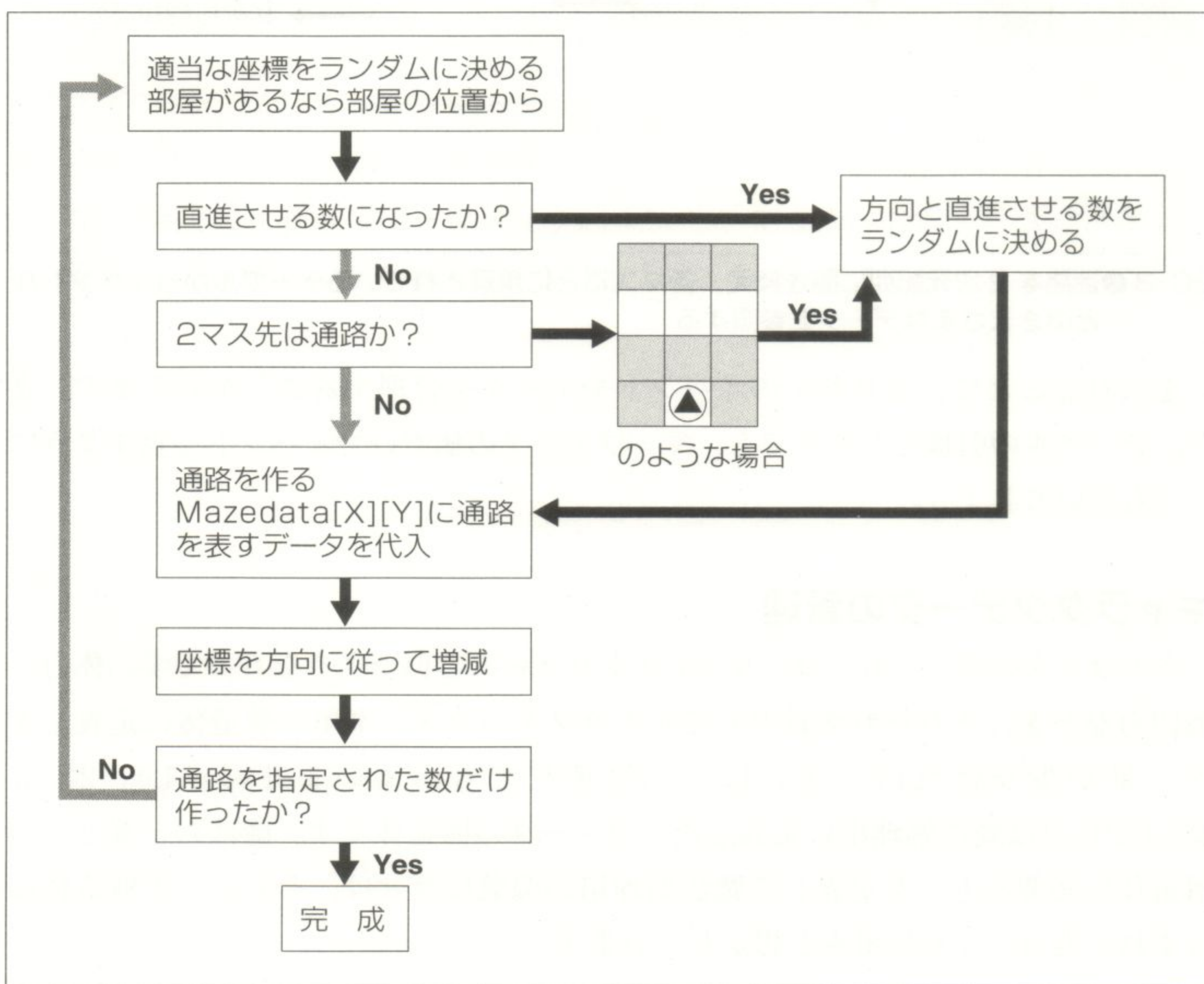



Fig. 2C-4 ●迷路の自動生成アルゴリズム

までの方向とは逆になったらその通路はいき止まりになります。最初に決めた直進させる数だけ通路を作ったら，そこで再び数と方向を決めます。そして，その数と方向で通路を作ります。この法則を繰り返すことで通路を伸ばしていきます。

ある程度生成が進むと，通路を作る方向に別の通路がぶつかることがあります。進行方向に別の通路があるようなら，直進させる数に関係なくその場で作る方向を新しく決めます。一筆書きのできるタイプ(枝道のない)の迷路を作りたいのなら，このときに別の通路と交差する方向は選ばないようにします。なお，進行方向に別の通路があるかどうかを知るには，直進方向の2マス先のところを調べなければいけません。通路を交差させずに手前で曲がるようにする場合は，通路と通路の間にある壁に厚さを与えなければならないからです。こうしないと仕切りとなる壁が作れません。

◆イベントを配置する

こうして通路ができたら迷路の完成です。法則に従って，紙へ通路を書いてみるとよりわかりやすいと思います。ゲームによっては通路とは別に部屋があります。アイテムやモンスタ/イベントも通路にばらまいておかなければなりません。これらはいずれも通路を作る最中に配置するようにします。たとえば「通路をある決められた数だけ作ったらアイテムを置く」という法則を作り，通路を作るたびにこのチェックをします。決められた数になったらアイテムを置いて，再び次の数を決めます。

この法則を決めるときに，「部屋にやたらとアイテムが多い」「キャラクタの最初の位置にアイテムが多く転がっている」などといった条件を作り，「部屋が作られたとき」「キャラクタの最初の位置が決まったとき」にアイテムを置くようにします。こうすれば，部屋と通路が繋がっていなかったり，アイテムがきちんと置かれなくなる，というミスを防ぐことができます。

List 2C-5，2C-6(P101)は，50×50のサイズで2次元配列に迷路を作る例です。Fig. 2C-5のような迷路が作られます。この例では2次元配列の範囲外になったときにも方向を変えています。

直進させる数 : 10
通路の数 : 600
部屋数 : 3

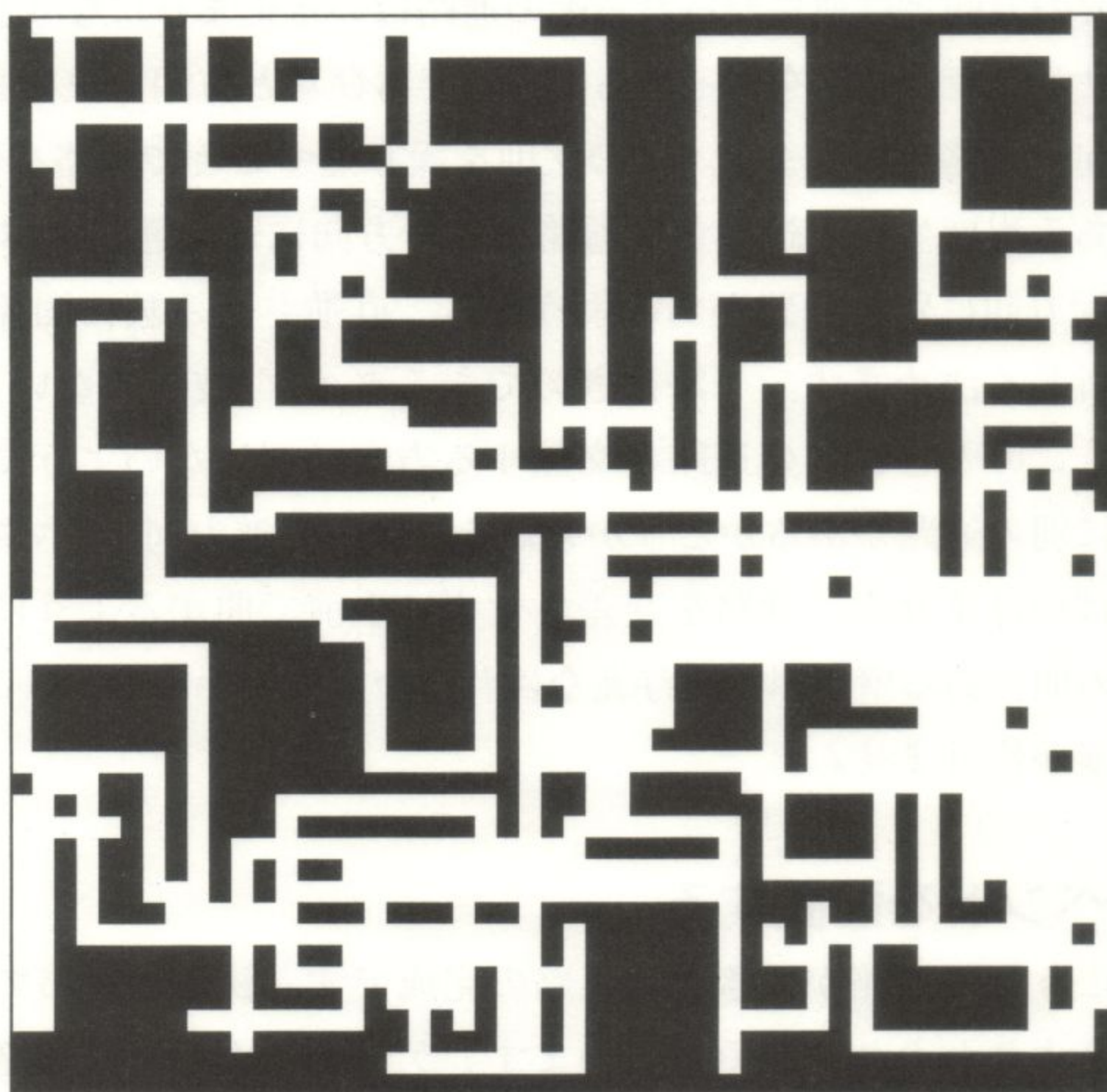



Fig. 2C-5●50×50のサイズで生成された迷路

迷路表示のアルゴリズム

プレイヤーの目の前にあたかも迷路が広がっているような表示は，一見しただけではどう描画しているのかわかりません。実は，遠近法を使って描かれたパーツを「奥行きに従って重ね合わせ」ているだけなのです。Fig. 2C-6はこの表示に使われるパーツです。この例では奥行きが3段階あります。これを使って少し説明しましょう。天井と地面だけがずっと広がっているパーツがあります。これを単独で表示すれば「周りに壁がない」状態です。これを背景として使い，そこへ壁のパーツを重ねていきます。

壁のパーツを選ぶには，迷路データをキャラクタの向いている方向へ奥行きの段階分だけ調べることが必要です。重ねる順番は「奥からいちばん手前へ」に向かって壁があるかどうか調べていき「縦方向の壁から横方向の壁」へと交互に置いていくのがよく使われている方法です。

◆壁を配置する

実際に壁を置く過程を見ていきましょう。まずいちばん奥にある4つの縦壁を調べます。縦壁を置くには「対応している位置に壁があるかどうか」，キャラクタ向いている方向に対して「壁の横方向に通路があるかどうか」の2つの条件を満たしていることが必要です。左側いちばん奥の壁の場合，キャラクタの位置から「奥へ3マス」,「左に1マス」のところに壁，同じく「奥へ3マス」が通路なら縦壁を置くことができます。縦壁が終わったら次は横壁です。横壁は「対応している位置に壁があるかどうか」「対応する位置の手前が通路かどうか」を条件として調べます(Fig. 2C-7)。

こうして手前まで壁のパーツを背景へと重ねていくのを繰り返したら，最後にキャラクタのいる「左右の縦壁」を調べます。これで完成です。奥からパーツを重ねるのでT字路のような「いちばん手前に縦壁」「いちばん奥に横壁」といった場合でも正しく表示できます。

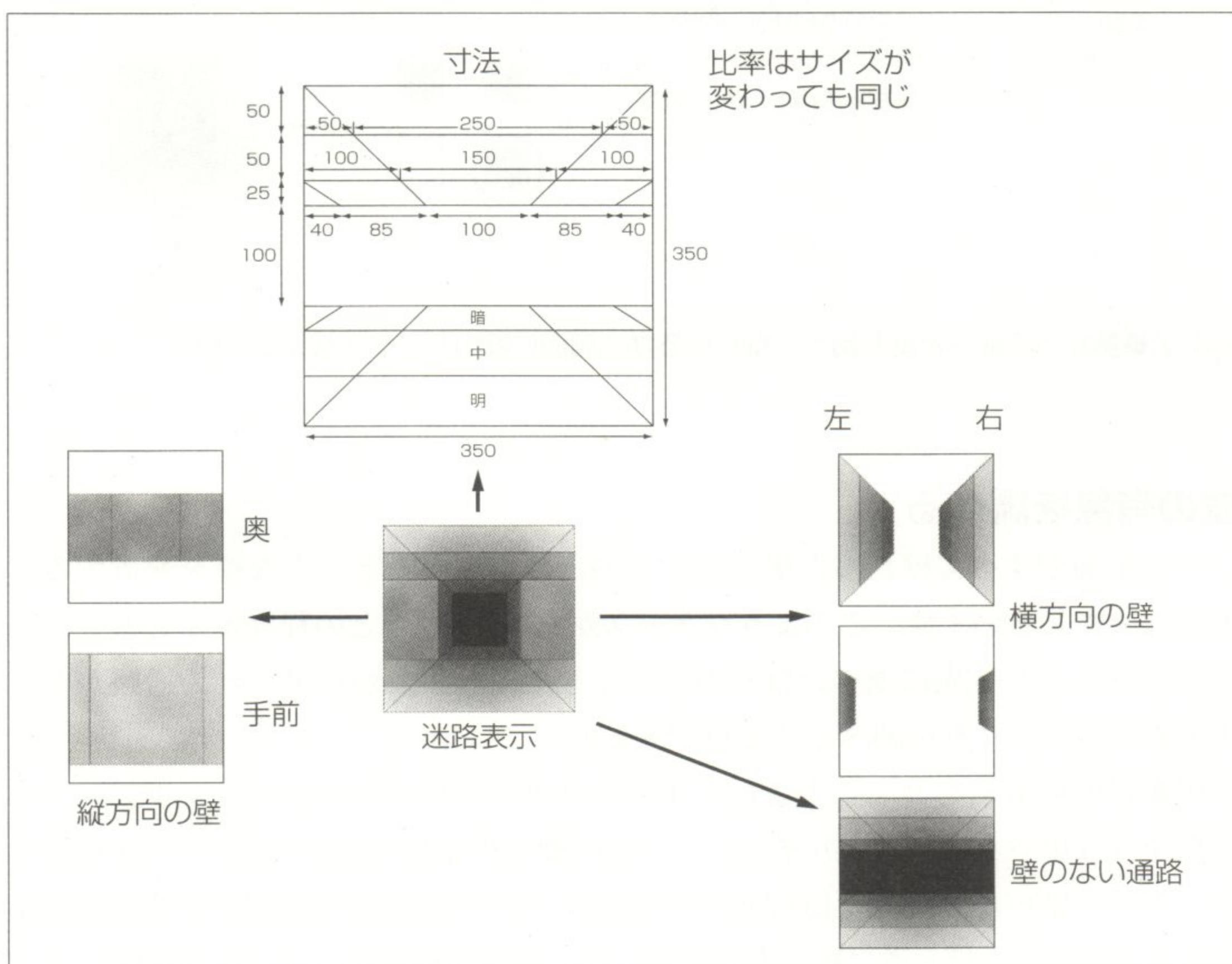



Fig. 2C-6 ●迷路表示に使うパーツ。縦横の壁をそれぞれ奥用と手前用のものを用意し，それらを組み合わせて遠近感のある画面を作り出す

背景を置く
縦方向に壁を置く
左 壁 通路 なら
右 通路 壁 なら
横方向に壁を置く
奥 壁
手前 通路
いちばん奥から手前へと繰り返す
いちばん手前にある縦方向の壁を置く
左右ともキャラクタの横に壁があるのなら
完成

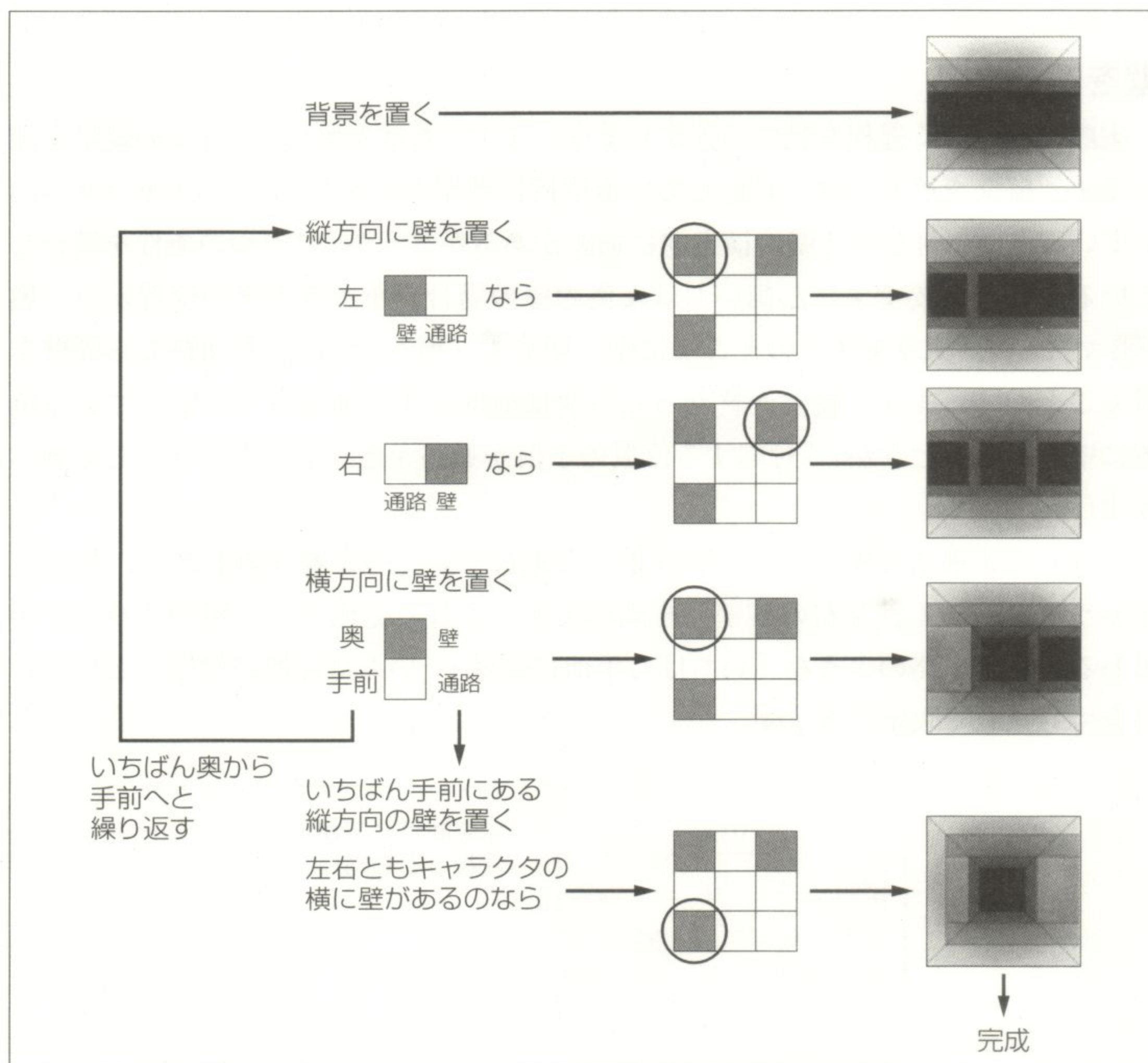



Fig. 2C-7 ●奥から手前へと壁を調べ，壁があるなら画面に壁のパーツを表示していく

◆壁の有無を調べる

もし手前がすべて壁だった場合は，奥のほうを調べるとムダな壁を配置することになってしまいます。このようなケースがあるため，「どの程度奥から壁を置くのを始めるか」を先に調べたほうがよいでしょう。手前から奥へと，通路がどこまであるかで，これを調べることができます。

実際に壁の有無を調べるときは，キャラクタの位置と向きからマップデータの座標を取り出さなければなりません。この座標を得るには2つのテーブルを使います。まず「壁がある位置」「通路がある位置」をそれぞれ1つのテーブルに集めます。たとえばいちばん奥に壁がある場合には，壁が「x＝0，y＝3」，通路が「x＝0，y＝2」の座標になければいけません。このときの座標は単に「どの位置だけ離

れているか」を示しているだけです。このテーブルを調べる壁の数だけ用意します。

もう1つのテーブルには，「現在の位置からキャラクタが向いている上下左右の方向に各1マス前はどれだけ離れているか」を集めておきます。キャラクタが北(上)を向いているとき，その1歩前はキャラクタの位置から「x = 0，y = -1」だけ離れています。西(左)なら「x = -1，y = 0」です。これを各向きに対して東西南北それぞれの各マス分だけ揃えます。

壁の有無を調べるときは，まず調べたい壁の位置を最初のテーブルから取り出して，次に2つ目のテーブルから「差」となるものを取り出し，これらをそれぞれ「掛」けます。これまでの例なら，

最初のテーブル　：　x=0, y=3
2つ目のテーブル　：　x=0, y=-1
掛ける　：　x=0×0, y=-1×3

で，結果として「x = 0，y = -3」が得られます。これは目的どおり「北方向に見て，3マス前にある地点」としてぴったりの座標となっています。こうしてテーブルを分割することで，効率的に座標を取り出すことができます。

List 2C-7，2C-8(P108)はC++BuilderとDelphiを使って，この迷路表示を作った例です。このルーチンはサンプルプログラムとして付属CD-ROMに収録してあります。

パーツの重ね合わせというと，ついつい四角いものやキャラクタだけを想像しがちですが，こうしたところにも応用されています。またこの奥行きを見せるトリック的な手法はほかのゲームでもよく使われています。

● プログラムの流れ

ダンジョンゲームのプログラムのおおまかな流れはFig. 2C-8のとおりです。まず迷路データを生成して，キャラクタの位置を決めたあと，画面を表示します。移動を指示するキーが押されたら，キャラクタの「方向」「座標」を更新して，それに対応するように表示なども変えてやります。イベントの判断もこのときに行います。イベント処理中にゲームの目的が達成されたら終了します。

アイテムはキャラクタ用に用意したパラメータに格納します。これはアイテム数分を範囲にしたテーブルにすると操作が簡単です。

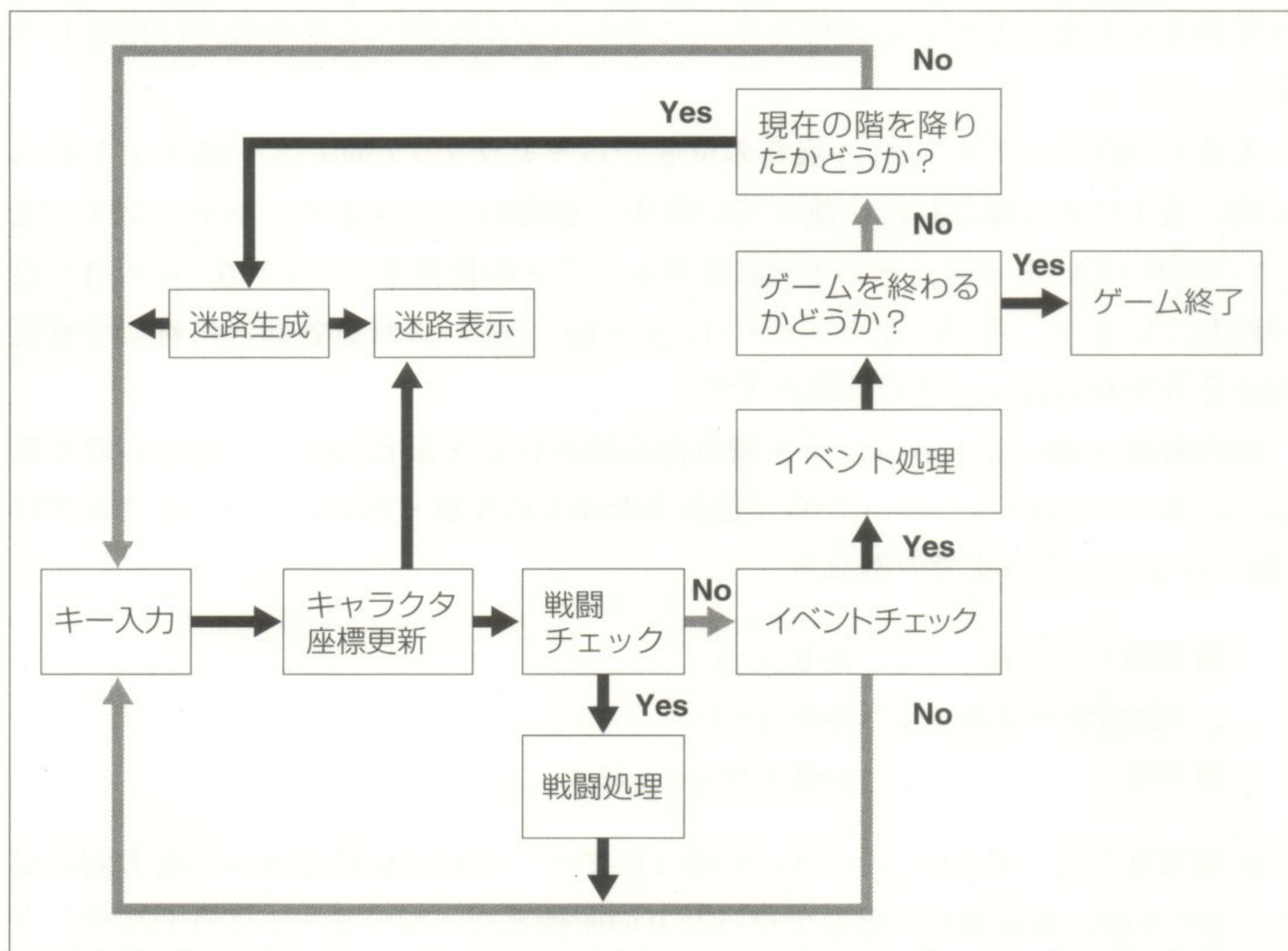



Fig. 2C-8 ●ダンジョンゲームのプログラムの流れ

モンスタの出現方法はいくつかあります。キャラクタが迷路内を歩いているときに突然出会うタイプでは，キャラクタが移動するたびに乱数を発生させて，それを基にモンスタと出会うかどうか決める方法が多いようです。あらかじめ決まった数のモンスタがいて，それが迷路内を動き回っているタイプになると，モンスタを移動させるためのルーチンが必要になります。これもけっきょくは移動方向などを乱数で決めるのですが，そのときに「一定時間は前にいたことのあるマスには移動しない」というように作ると動きがよくなります。

● サンプルゲームの遊び方

プレイヤーは主人公ミストに勝手に召喚され，ミストといっしょに迷路の中をさまよい歩きます。迷路の途中には「魔法の言葉」が詰まったボールが落ちています。これを見つけて多くの魔法の言葉を集めて，自分だけの魔法を作らなければなりません。「食べ物」「武器」といったアイテムはありません。そのかわり魔法をうまく作り出して戦闘や体力回復に使いながら，謎が待ちかまえる迷宮の奥へ

と進んでいきます。

迷路は自動生成されます。そのため，必ずしもマップ上全体に通路があるわけではありません。ある程度アイテムを得たら，すぐにその階から脱出することを考えたほうが早くゲームを終わらせることができます。敵との闘いは普通のRPGとほぼ同じです。魔法の強さは技そのものの力に加えて「体力」「魔力」と比例するようにしました。たとえ大きな力を持った魔法でも，唱える人の魔力が弱ければそれ相応の力しか出すことができません。また魔法のなかには体力を消耗するものがあるので，このあたりも考えながらプレイすることが必要です。体力は歩いていると少しずつ回復します。

このゲームでは，シーンごとに声が出ます。声を収録させていただいた藤崎真美さんと，協力していただいた酒井伸和さんに感謝します。

List 2C-1 ●迷路データ（Delphi）

```
{ 定数宣言 }
const
   CellLimit = 16;
   MazeWidthLimit = 50;
    MazeHeightLimit = 50;

{ 型宣言 }
type
   TCell = record          { セルの定義        }
        Wall: integer;     { 壁かどうか        }
        Event: integer;    { イベントフラグ    }
        Move: integer;     { 移動可能フラグ    }
    end;

{ 変数宣言 }
var
    { セルテーブル }
    CellTbl : array[0..CellLimit] of TCell;

    { 迷路データ }
    Mazedata : array[0..MazeWidthLimit,
                 0..MazeHeightLimit] of byte;
```

List 2C-2●迷路データ(C/C++)

```
/* 定数宣言 */
#define CellLimit       16
#define MazeWidthLimit  50
#define MazeHeightLimit 50

/* 型宣言 */
typedef struct {        /* セルの定義        */
    int Wall;           /* 壁かどうか        */
    int Event;          /* イベントフラグ    */
    int Move;           /* 移動可能フラグ    */
} TCell;

/* 変数宣言 */
/* セルテーブル */
TCell CellTbl[CellLimit + 1];

/* 迷路データ */
unsigned char Mazedata[MazeWidthLimit + 1]
                           [MazeHeightLimit + 1];
```

List 2C-3●魔法処理用のパラメータ(Delphi)

```
{ 型宣言 }
type
    TMagicParam = record
        Element: integer;    { 魔法のカテゴリ }
        Category: integer;   { 種類 }
        Power: integer;      { 魔法力 }
        HP: integer;         { 体力の変化 }
        MP: integer;         { 魔力の変化 }
        Flag: integer;       { フラグ }
    end;

    TMagicSrcParam = record
        Category: integer;   { 魔法の種類 }
        HP: integer;         { 使う人間の体力 }
        MP: integer;         { 使う人間の魔力 }
        Enemy: integer;      { 敵の種類 }
        EnemyHP: integer;    { 敵の体力 }
        EnemyMP: integer;    { 敵の魔力 }
        OrgEnemyMP,OrgEnemyHP: integer;
    end;
```

List 2C-4 ●魔法処理用のパラメータ(C/C++)

```
typedef struct {
    int Element;     //魔法のカテゴリ
    int Category;    //種類
    int Power;       //魔法力
    int HP;          //体力の変化
    int MP;          //魔力の変化
    int Flag;        //フラグ
} TMagicParam;

typedef struct {
    int Category;    //魔法の種類
    int HP;          //使う人間の体力
    int MP;          //使う人間の魔力
    int Enemy;       //敵の種類
    int EnemyHP;     //敵の体力
    int EnemyMP;     //敵の魔力
    int OrgEnemyMP,OrgEnemyHP;
} TMagicSrcParam;
```

List 2C-5 ●迷路生成(Delphi)

```
{ 型宣言 }
type
    TPoint = record
        x,y: integer;
    end;

{ 定数宣言 }
const
    CellRoute = 0;
    CellWall = 1;
    MazeWidthLimit = 50;
    MazeHeightLimit = 50;
    RoomXLimit = 20;
    RoomYLimit = 20;

    RandomStart = -1;

    DirectionEast = 0;
    DirectionWest = 1;
    DirectionSouth = 2;
    DirectionNorth = 3;
    DirectionLimit = 4;
```

▼

List 2C-5

```
    DifTbl: array[0..DirectionLimit-1] of TPoint = (
        (X: 1;  Y: 0),
        (X: -1; Y: 0),
        (X: 0;  Y: 1),
        (X: 0;  Y: -1)
    );

{ 変数宣言 }
var
    { 迷路データ }
    Mazedata: array[0..MazeWidthLimit,
                    0..MazeHeightLimit] of byte;


(* ------------------------------------------------ *)
{ 迷路初期化 }
procedure InitMaze;
var
   x, y: integer;
begin
    for y := 0 to MazeHeightLimit-1 do begin
        for x := 0 to MazeWidthLimit-1 do begin
            Mazedata[x][y] := CellWall;
        end;
    end;
end;

{ 部屋生成 }
procedure MakeRoom(LimitX,
            LimitY: integer; var StartPoint: TPoint);
var
   RoomX, RoomY, MazeOfsX, MazeOfsY, x, y: integer;
begin
    RoomX := Random(LimitX);
    RoomY := Random(LimitY);
    if RoomX = 0 then
        RoomX := 2;
    if RoomY = 0 then
        RoomY := 2;
    MazeOfsX := Random(MazeWidthLimit - RoomX - 1);
    MazeOfsY := Random(MazeHeightLimit - RoomY - 1);
    for y := 0 to RoomY-1 do begin
        for x := 0 to RoomX-1 do begin
            Mazedata[MazeOfsX + x]
                    [MazeOfsY + y] := CellRoute;
        end;
    end;
    StartPoint.X := MazeOfsX + (RoomX div 2);
    StartPoint.Y := MazeOfsY + (RoomY div 2);
```

```
end;

{ 範囲チェック }
function CheckRange(RouteX, RouteY: integer): boolean;
begin
    if (RouteX < 0) or (RouteX >= MazeWidthLimit) or
            (RouteY < 0) or (RouteY >= MazeHeightLimit)
                                              then begin
        result := True;
        exit;
    end;
    result := False;
end;

{ 通路を作る方向にほかの通路があるかどうか }
function CheckRoute(RouteX, RouteY,
                       Direction: integer): boolean;
var
   i: integer;
begin
    for i := 0 to 1 do begin
        RouteX := RouteX + DifTbl[Direction].x;
        RouteY := RouteY + DifTbl[Direction].y;
    end;
    if CheckRange(RouteX, RouteY) then begin
        result := True;
        exit;
    end;
    if Mazedata[RouteX][RouteY] = CellRoute then begin
        result := True;
        exit;
    end;
    result := False;
end;

{ 通路生成 }
procedure MakeRoute(LimitFlat, LimitRoute,
                              RouteX, RouteY: integer);
var
  i, OldRouteX, OldRouteY, FlatCount, Direction: integer;
begin
    if RouteX = RandomStart then
        RouteX := Random(MazeWidthLimit - 1);
    if RouteY = RandomStart then
        RouteY := Random(MazeHeightLimit - 1);
    FlatCount := 0;
    for i := 0 to LimitRoute-1 do begin
        if (FlatCount = 0) or CheckRoute(RouteX,
                             RouteY, Direction) then begin
            FlatCount := Random(LimitFlat);
```

List 2C-5

```
          Direction := Random(DirectionLimit);
        end;
        OldRouteX := RouteX;
        OldRouteY := RouteY;
        RouteX := RouteX + DifTbl[Direction].x;
        RouteY := RouteY + DifTbl[Direction].y;
        if CheckRange(RouteX, RouteY) then begin
            RouteX := OldRouteX;
            RouteY := OldRouteY;
            FlatCount := Random(LimitFlat);
            repeat
                Direction := Random(DirectionLimit);
            until CheckRange(RouteX + DifTbl[Direction].x,
                    RouteY + DifTbl[Direction].y);
        end;
        Mazedata[RouteX][RouteY] := CellRoute;
        if FlatCount <> 0 then
             Dec(FlatCount);
    end;
end;


(* -------------------------------------------- *)
{ 迷路生成 }
procedure MakeMaze(LimitFlat, LimitRoute,
                                   LimitRoom: integer);
var
   i: integer;
   RouteStart: TPoint;
begin
    InitMaze;
    if LimitRoom <> 0 then begin
        for i := 0 to Random(LimitRoom) do begin
            MakeRoom(RoomXLimit, RoomYLimit, RouteStart);
            MakeRoute(LimitFlat, LimitRoute,
                            RouteStart.X, RouteStart.Y);
        end;
    end;
    MakeRoute(LimitFlat, LimitRoute,
                            RandomStart, RandomStart);
end;
```

List 2C-6●迷路生成(C/C++)

```
/* 型宣言 */
typedef struct {
    int x;
    int y;
} TPoint;

/* 定数宣言 */
#define CellRoute          0
#define CellWall           1
#define MazeWidthLimit    50
#define MazeHeightLimit   50
#define RoomXLimit        20
#define RoomYLimit        20

#define RandomStart    -1

#define DirectionEast  0
#define DirectionWest  1
#define DirectionSouth 2
#define DirectionNorth 3
#define DirectionLimit 4

static TPoint DifTbl[DirectionLimit + 1] = {
    1,  0,
    -1, 0,
    0,  1,
    0,  -1,
};

/* 変数宣言 */
/* 迷路データ */
unsigned char Mazedata[MazeWidthLimit + 1]
                           [MazeHeightLimit + 1];


/* ----------------------------------------------- */
/* 迷路初期化 */
void InitMaze(void)
{
    int x, y;

    for (y = 0; y < MazeHeightLimit; y++) {
        for (x = 0; x < MazeWidthLimit; x++) {
            Mazedata[x][y] = CellWall;
        }
    }
}
```

List 2C-6

```
/＊ 部屋生成 ＊/
void MakeRoom(int LimitX, int LimitY,
                                    TPoint ＊StartPoint)
{
    int RoomX, RoomY, MazeOfsX, MazeOfsY, x, y;

    RoomX = random(LimitX);
    RoomY = random(LimitY);
    if (RoomX == 0)
        RoomX = 2;
    if (RoomY == 0)
        RoomY = 2;
    MazeOfsX = random(MazeWidthLimit - RoomX - 1);
    MazeOfsY = random(MazeHeightLimit - RoomY - 1);
    for (y = 0; y < RoomY; y++) {
        for (x = 0; x < RoomX; x++) {
            Mazedata[MazeOfsX + x]
                               [MazeOfsY + y] = CellRoute;
        }
    }
    StartPoint->x = MazeOfsX + (RoomX / 2);
    StartPoint->y = MazeOfsY + (RoomY / 2);
}

/＊ 範囲チェック ＊/
int CheckRange(int RouteX, int RouteY)
{
    if ((RouteX < 0) || (RouteX >= MazeWidthLimit) ||
        (RouteY < 0) || (RouteY >= MazeHeightLimit)) {
        return True;
    }
    return False;
}

/＊ 通路を作る方向にほかの通路があるかどうか ＊/
int CheckRoute(int RouteX, int RouteY, int Direction)
{
    int i;

    for (i = 0; i < 2; i++) {
        RouteX += DifTbl[Direction].x;
        RouteY += DifTbl[Direction].y;
    }
    if (CheckRange(RouteX, RouteY)) {
        return True;
    }
    if (Mazedata[RouteX][RouteY] == CellRoute) {
        return True;
    }
    return False;
```

```
}

/* 通路生成 */
void MakeRoute(int LimitFlat, int LimitRoute,
                              int RouteX, int RouteY)
{
    int i, OldRouteX, OldRouteY, FlatCount, Direction;

    if (RouteX == RandomStart)
        RouteX = random(MazeWidthLimit);
    if (RouteY == RandomStart)
        RouteY = random(MazeHeightLimit);
    FlatCount = 0;
    for (i = 0; i < LimitRoute; i++) {
        if ((FlatCount == 0) || (CheckRoute(RouteX,
                                  RouteY, Direction))) {
            FlatCount = random(LimitFlat);
            Direction = random(DirectionLimit);
        }
        OldRouteX = RouteX;
        OldRouteY = RouteY;
        RouteX = RouteX + DifTbl[Direction].x;
        RouteY = RouteY + DifTbl[Direction].y;
        if (CheckRange(RouteX, RouteY)) {
            RouteX = OldRouteX;
            RouteY = OldRouteY;
            FlatCount = random(LimitFlat);
            do {
                Direction = random(DirectionLimit);
            } while(CheckRange(RouteX + DifTbl[Direction].x,
                           RouteY + DifTbl[Direction].y));
        }
        Mazedata[RouteX][RouteY] = CellRoute;
        if (FlatCount >= 0)
             FlatCount--;
    }
}


/* ------------------------------------------------ */
/* 迷路生成 */
void MakeMaze(int LimitFlat, int LimitRoute, int LimitRoom)
{
    int i;
    TPoint RouteStart;

    InitMaze();
    if (LimitRoom != 0) {
        for (i = 0; i < random(LimitRoom); i++) {
            MakeRoom(RoomXLimit, RoomYLimit, &RouteStart);
```

List 2C-6

```
            MakeRoute(LimitFlat, LimitRoute,
                                  RouteStart.x, RouteStart.y);
        }
    }
    MakeRoute(LimitFlat, LimitRoute,
                               RandomStart, RandomStart);
}
```

List 2C-7●迷路表示（Delphi）

```
{ 型宣言 }
type
    TPoint = record
        x, y: integer;
    end;

    TRect = record
        Left, Top, Right, Bottom: integer;
    end;

    TDirectionDifOfs = record
        Up: TPoint;      { 上方向 }
        Down: TPoint;    { 下方向 }
        Right: TPoint;   { 右方向 }
        Left: TPoint;    { 左方向 }
    end;

    TWallOfs = record
        BmpOfs: integer;     { ビットマップの種類 }
        SpOfs: integer;      { スプライトの位置 }
        WallArray : array[0..4-1] of byte;  { 壁の位置 }
        RouteArray : array[0..4-1] of byte; { 通路の位置 }
    end;

{ 定数宣言 }
const
    ArrayLeft   = 0;
    ArrayTop    = 1;
    ArrayRight  = 2;
    ArrayBottom = 3;

    BitmapHeightWallTopLeft      = 0;
    BitmapHeightWallTopRight     = 1;
    BitmapHeightWallMidLeft      = 2;
    BitmapHeightWallMidRight     = 3;
    BitmapHeightWallLowLeft      = 4;
```

```
    BitmapHeightWallLowRight    = 5;
    BitmapHeightWallUnderLeft   = 6;
    BitmapHeightWallUnderRight  = 7;
    BitmapWidthWallTopLeft      = 8;
    BitmapWidthWallTopCenter    = 9;
    BitmapWidthWallTopRight     = 10;
    BitmapWidthWallMidLeft      = 11;
    BitmapWidthWallMidCenter    = 12;
    BitmapWidthWallMidRight     = 13;
    BitmapWallLimit             = 14;
    BitmapNoWall                = 14;
    BitmapAllWallLimit          = 15;

    DirectionEast = 0;
    DirectionWest = 1;
    DirectionSouth = 2;
    DirectionNorth = 3;
    DirectionLimit = 4;

{ 東西南北で上下左右1マス先の位置を取り出すためのテーブル }
  DirectionOfsTbl: array[0..DirectionLimit-1]
                                      of TDirectionDifOfs = (
    (Up: (X: 1; Y: 0); Down: (X:-1; Y: 0); Right: (X: 0; Y: 1);
                                        Left: (X:0;  Y:-1)),
    (Up: (X:-1; Y: 0); Down: (X: 1; Y: 0); Right: (X: 0; Y:-1);
                                        Left: (X:0;  Y: 1)),
    (Up: (X: 0; Y: 1); Down: (X: 0; Y:-1); Right: (X:-1; Y: 0);
                                        Left: (X: 1; Y: 0)),
    (Up: (X: 0; Y:-1); Down: (X: 0; Y: 1); Right: (X: 1; Y: 0);
                                        Left: (X:-1; Y: 0))
)

{ 壁と通路の位置を取り出すためのテーブル }
  WallOfsTbl: array[0..BitmapWallLimit-1] of TWallOfs = (
    (BmpOfs: BitmapHeightWallUnderLeft;  SpOfs: 0;
             WallArray: (2, 2, 0, 0); RouteArray: (1, 2, 0, 0)),
    (BmpOfs: BitmapHeightWallUnderRight; SpOfs: 1;
             WallArray: (0, 2, 2, 0); RouteArray: (0, 2, 1, 0)),
    (BmpOfs: BitmapHeightWallLowLeft;    SpOfs: 2;
             WallArray: (1, 2, 0, 0); RouteArray: (0, 2, 0, 0)),
    (BmpOfs: BitmapHeightWallLowRight;   SpOfs: 3;
             WallArray: (0, 2, 1, 0); RouteArray: (0, 2, 0, 0)),
    (BmpOfs: BitmapWidthWallMidLeft;     SpOfs: 4;
             WallArray: (1, 2, 0, 0); RouteArray: (1, 1, 0, 0)),
    (BmpOfs: BitmapWidthWallMidCenter;   SpOfs: 5;
             WallArray: (0, 2, 0, 0); RouteArray: (0, 1, 0, 0)),
    (BmpOfs: BitmapWidthWallMidRight;    SpOfs: 6;
             WallArray: (0, 2, 1, 0); RouteArray: (0, 1, 1, 0)),
    (BmpOfs: BitmapHeightWallMidLeft;    SpOfs: 7;
             WallArray: (1, 1, 0, 0); RouteArray: (0, 1, 0, 0)),
```

List 2C-7

```
    (BmpOfs: BitmapHeightWallMidRight;    SpOfs: 8;
               WallArray: (0, 1, 1, 0); RouteArray: (0, 1, 0, 0)),
    (BmpOfs: BitmapWidthWallTopLeft;      SpOfs: 9;
               WallArray: (1, 1, 0, 0); RouteArray: (1, 0, 0, 0)),
    (BmpOfs: BitmapWidthWallTopCenter;  SpOfs: 10;
               WallArray: (0, 1, 0, 0); RouteArray: (0, 0, 0, 0)),
    (BmpOfs: BitmapWidthWallTopRight;   SpOfs: 11;
               WallArray: (0, 1, 1, 0); RouteArray: (0, 0, 1, 0)),
    (BmpOfs: BitmapHeightWallTopLeft;   SpOfs: 12;
               WallArray: (1, 0, 0, 0); RouteArray: (0, 0, 0, 0)),
    (BmpOfs: BitmapHeightWallTopRight;  SpOfs: 13;
               WallArray: (0, 0, 1, 0); RouteArray: (0, 0, 0, 0))
  );


{ ---------------------------------------------------------------- }
{ 壁かどうか調べる位置をキャラクタがいる地点からマップ座標へ変換 }
procedure TMaze.GetWallOfs(Ofs: integer; var WallX, WallY: integer);
var
    TmpRect: TRect;
begin
    TmpRect := Rect(WallOfsTbl[Ofs].WallArray[ArrayLeft],
                    WallOfsTbl[Ofs].WallArray[ArrayTop],
                    WallOfsTbl[Ofs].WallArray[ArrayRight],
                    WallOfsTbl[Ofs].WallArray[ArrayBottom]);
    if TmpRect.Right <> 0 then begin
        WallX := TmpRect.Right * DirectionDifOfsTbl
                                    [FCharacterDirection].Right.x;
        WallY := TmpRect.Right * DirectionDifOfsTbl
                                    [FCharacterDirection].Right.y;
    end else begin
        WallX := TmpRect.Left * DirectionDifOfsTbl
                                    [FCharacterDirection].Left.x;
        WallY := TmpRect.Left * DirectionDifOfsTbl
                                  [FCharacterDirection].Left.y;
    end;
    if WallX = 0 then
        WallX := TmpRect.Top * DirectionDifOfsTbl
                                  [FCharacterDirection].Up.x;
    if WallY = 0 then
        WallY := TmpRect.Top * DirectionDifOfsTbl
                                  [FCharacterDirection].Up.y;
    WallX := WallX + FCharacterX;  { 現在のキャラクタ X 座標 }
    WallY := WallY + FCharacterY;  { 現在のキャラクタ Y 座標 }
end;

{ 通路かどうか調べる位置をキャラクタがいる地点からマップ座標へ変換 }
procedure TMaze.GetRouteOfs(Ofs: integer; var RouteX, RouteY: integer);
var
    TmpRect: TRect;
```

```
begin
    TmpRect := Rect(WallOfsTbl[Ofs].RouteArray[ArrayLeft],
                    WallOfsTbl[Ofs].RouteArray[ArrayTop],
                    WallOfsTbl[Ofs].RouteArray[ArrayRight],
                    WallOfsTbl[Ofs].RouteArray[ArrayBottom]);

    if (TmpRect.Right = 0) and (TmpRect.Left = 0) and
                                (TmpRect.Top = 0) then begin
        RouteX := -1;
        RouteY := -1;
        exit;
    end;
    if TmpRect.Right <> 0 then begin
        RouteX := TmpRect.Right * DirectionDifOfsTbl
                          [FCharacterDirection].Right.x;
        RouteY := TmpRect.Right * DirectionDifOfsTbl
                          [FCharacterDirection].Right.y;
    end else begin
        RouteX := TmpRect.Left * DirectionDifOfsTbl
                          [FCharacterDirection].Left.x;
        RouteY := TmpRect.Left * DirectionDifOfsTbl
                          [FCharacterDirection].Left.y;
    end;
    if RouteX = 0 then
        RouteX := TmpRect.Top * DirectionDifOfsTbl
                          [FCharacterDirection].Up.x;
    if RouteY = 0 then
        RouteY := TmpRect.Top * DirectionDifOfsTbl
                          [FCharacterDirection].Up.y;
    RouteX := RouteX + FCharacterX;  { 現在のキャラクタ X 座標 }
    RouteY := RouteY + FCharacterY;  { 現在のキャラクタ Y 座標 }
}

{ 迷路データを描画 }
procedure TMaze.Draw(x, y, direction: integer;
                              Sp: TSp, ScreenBitmap: TBitmap);
var
    WallX, WallY: integer;    { 壁でなければならない地点 }
    RouteX, RouteY: integer;  { 通路でなければならない地点 }
    i: integer;
begin
    { 壁がない状態を描画 }
    ScreenBitmap.Canvas.Draw(0, 0, BitmapList[BitmapNoWall]);
    for i := 0 to BitmapWallLimit-1 do begin
        { 取りあえずいったん対応するパーツを非表示にする }
        Sp.ChangeEnabled(WallOfsTbl[i].SpOfs, False);
        { 壁かどうかを調べる位置を出す }
        GetWallOfs(i, WallX, WallY);
        { 通路かどうかを調べる位置を出す }
        GetRouteOfs(i, RouteX, RouteY);
```

List 2C-7

```
        if (RouteX <> -1) and (RouteY <> -1) then begin
            { 通路であってほしいところが壁ならふたたびループへ }
            if isWall(RouteX, RouteY) then
                Continue;
        end;
        if isWall(WallX, WallY) then begin
            { 条件があっていればそのスプライトを表示可能に }
            Sp.ChangeEnabled(WallOfsTbl[i].SpOfs, True);
        end;
    }
    Sp.View(ScreenBitmap);
end;
```

List 2C-8●迷路表示(C/C++)

```
/* 型宣言 */
typedef struct {
    int x, y;
} TPoint;

typedef struct {
    int Left, Top, Right, Bottom;
} TRect;

typedef struct {
    TPoint  Up;     // 上方向
    TPoint  Down;   // 下方向
    TPoint  Right;  // 右方向
    TPoint  Left;   // 左方向
} TDirectionDifOfs;

typedef struct {
    int BmpOfs;                 // ビットマップの種類
    int SpOfs;                  // スプライトの位置
    short WallArray[4];         // 壁の位置
    short RouteArray[4];        // 通路の位置
} TWallOfs;

/* 定数宣言 */
#define ArrayLeft       0
#define ArrayTop        1
#define ArrayRight      2
#define ArrayBottom     3

#define BitmapHeightWallTopLeft         0
#define BitmapHeightWallTopRight        1
```

```
#define BitmapHeightWallMidLeft        2
#define BitmapHeightWallMidRight       3
#define BitmapHeightWallLowLeft        4
#define BitmapHeightWallLowRight       5
#define BitmapHeightWallUnderLeft      6
#define BitmapHeightWallUnderRight     7
#define BitmapWidthWallTopLeft         8
#define BitmapWidthWallTopCenter       9
#define BitmapWidthWallTopRight        10
#define BitmapWidthWallMidLeft         11
#define BitmapWidthWallMidCenter       12
#define BitmapWidthWallMidRight        13
#define BitmapWallLimit                14
#define BitmapNoWall                   14
#define BitmapAllWallLimit             15

#define DirectionEast     0
#define DirectionWest     1
#define DirectionSouth    2
#define DirectionNorth    3
#define DirectionLimit    4

/* 東西南北で上下左右１マス先の位置を取り出すためのテーブル */
TDirectionDifOfs DirectionDifOfsTbl[] = {
    {{ 1,  0}, {-1,  0}, { 0,  1}, {0,  -1}},
    {{-1,  0}, { 1,  0}, { 0, -1}, {0,   1}},
    {{ 0,  1}, { 0, -1}, {-1,  0}, { 1,  0}},
    {{ 0, -1}, { 0,  1}, { 1,  0}, {-1,  0}},
};

/* 壁と通路の位置を取り出すためのテーブル */
TWallOfs WallOfsTbl[] = {
    {BitmapHeightWallUnderLeft,  0, {2, 2, 0, 0}, {1, 2, 0, 0}},
    {BitmapHeightWallUnderRight, 1, {0, 2, 2, 0}, {0, 2, 1, 0}},
    {BitmapHeightWallLowLeft,    2, {1, 2, 0, 0}, {0, 2, 0, 0}},
    {BitmapHeightWallLowRight,   3, {0, 2, 1, 0}, {0, 2, 0, 0}},
    {BitmapWidthWallMidLeft,     4, {1, 2, 0, 0}, {1, 1, 0, 0}},
    {BitmapWidthWallMidCenter,   5, {0, 2, 0, 0}, {0, 1, 0, 0}},
    {BitmapWidthWallMidRight,    6, {0, 2, 1, 0}, {0, 1, 1, 0}},
    {BitmapHeightWallMidLeft,    7, {1, 1, 0, 0}, {0, 1, 0, 0}},
    {BitmapHeightWallMidRight,   8, {0, 1, 1, 0}, {0, 1, 0, 0}},
    {BitmapWidthWallTopLeft,     9, {1, 1, 0, 0}, {1, 0, 0, 0}},
    {BitmapWidthWallTopCenter,  10, {0, 1, 0, 0}, {0, 0, 0, 0}},
    {BitmapWidthWallTopRight,   11, {0, 1, 1, 0}, {0, 0, 1, 0}},
    {BitmapHeightWallTopLeft,   12, {1, 0, 0, 0}, {0, 0, 0, 0}},
    {BitmapHeightWallTopRight,  13, {0, 0, 1, 0}, {0, 0, 0, 0}},
};

//-------------------------------------------------------------------
```

List 2C-8

```
// 壁かどうか調べる位置をキャラクタがいる地点からマップ座標へ変換
void __fastcall TMaze::GetWallOfs(int Ofs, int &WallX, int &WallY)
{
    TRect TmpRect = Rect(WallOfsTbl[Ofs].WallArray[ArrayLeft],
                         WallOfsTbl[Ofs].WallArray[ArrayTop],
                         WallOfsTbl[Ofs].WallArray[ArrayRight],
                         WallOfsTbl[Ofs].WallArray[ArrayBottom]);

    if (TmpRect.Right != 0) {
        WallX = TmpRect.Right * DirectionDifOfsTbl
                                    [FCharacterDirection].Right.x;
        WallY = TmpRect.Right * DirectionDifOfsTbl
                                    [FCharacterDirection].Right.y;
    } else {
        WallX = TmpRect.Left * DirectionDifOfsTbl
                                    [FCharacterDirection].Left.x;
        WallY = TmpRect.Left * DirectionDifOfsTbl
                                    [FCharacterDirection].Left.y;
    }
    if (WallX == 0)
        WallX = TmpRect.Top * DirectionDifOfsTbl
                                    [FCharacterDirection].Up.x;
    if (WallY == 0)
        WallY = TmpRect.Top * DirectionDifOfsTbl
                                    [FCharacterDirection].Up.y;
    WallX += FCharacterX; // 現在のキャラクタ X 座標
    WallY += FCharacterY; // 現在のキャラクタ Y 座標
}

// 通路かどうか調べる位置をキャラクタがいる地点からマップ座標へ変換
void __fastcall TMaze::GetRouteOfs(int Ofs, int &RouteX, int &RouteY)
{
    TRect TmpRect = Rect(WallOfsTbl[Ofs].RouteArray[ArrayLeft],
                         WallOfsTbl[Ofs].RouteArray[ArrayTop],
                         WallOfsTbl[Ofs].RouteArray[ArrayRight],
                         WallOfsTbl[Ofs].RouteArray[ArrayBottom]);

    if ((TmpRect.Right == 0) && (TmpRect.Left == 0) &&
                                    (TmpRect.Top == 0)) {
        RouteX = -1;
        RouteY = -1;
        return;
    }
    if (TmpRect.Right != 0) {
        RouteX = TmpRect.Right * DirectionDifOfsTbl
                                    [FCharacterDirection].Right.x;
        RouteY = TmpRect.Right * DirectionDifOfsTbl
                                    [FCharacterDirection].Right.y;
    } else {
        RouteX = TmpRect.Left * DirectionDifOfsTbl
```

```
                                [FCharacterDirection].Left.x;
        RouteY = TmpRect.Left * DirectionDifOfsTbl
                                [FCharacterDirection].Left.y;
    }
    if (RouteX == 0)
        RouteX = TmpRect.Top * DirectionDifOfsTbl
                                [FCharacterDirection].Up.x;
    if (RouteY == 0)
        RouteY = TmpRect.Top * DirectionDifOfsTbl
                                [FCharacterDirection].Up.y;
    RouteX += FCharacterX; // 現在のキャラクタ X 座標
    RouteY += FCharacterY; // 現在のキャラクタ Y 座標
}

// 迷路データを描画
void __fastcall TMaze::Draw(int x, int y, int direction,
                            TSp *Sp, TBitmap *ScreenBitmap)
{
    int WallX, WallY;     // 壁でなければならない地点
    int RouteX, RouteY;   // 通路でなければならない地点

    // 壁がない状態を描画
    ScreenBitmap->Canvas->Draw(0, 0, BitmapList[BitmapNoWall]);
    for (int i = 0; i < BitmapWallLimit; i++) {
        //取りあえずいったん対応するパーツを非表示にする
        Sp->ChangeEnabled(WallOfsTbl[i].SpOfs, False);
        //壁かどうかを調べる位置を出す
        GetWallOfs(i, WallX, WallY);
        //通路かどうかを調べる位置を出す
        GetRouteOfs(i, RouteX, RouteY);
        if ((RouteX != -1) && (RouteY != -1)) {
            // 通路であってほしいところが壁ならふたたびループへ
            if (isWall(RouteX, RouteY))
                continue;
        }
        if (isWall(WallX, WallY)) {
            //条件があっていればそのスプライトを表示可能に
            Sp->ChangeEnabled(WallOfsTbl[i].SpOfs, True);
        }
    }
    Sp->View(ScreenBitmap);
}
```

Section ④ シミュレーションゲーム

シミュレーションゲームの中身はどうなっているのでしょうか？　シミュレーションゲームで使われる「パラメータ」というものを通して，そのアルゴリズムに触れてみようと思います

● もの作る喜びを味わう

ものを作るというのは何ごとにも変えがたい喜びです。できあがっていく過程を眺めるだけでも楽しめます。でも「作ったものを成長させる」というのはなかなか体験できることではありません。植物や動物を育てていくのが感覚的に近いのかもしれませんが，遺伝子レベルで0から構築するというのはとても無理です。これは神様だけができることかもしれません。でも私たちにはコンピュータがあります。コンピュータを使えば，環境やキャラクタを擬似的に作り出し，それがどのように成長するのかが，誰にでも見ることができます。これこそコンピュータが与えてくれた新しい種類の喜びでしょう。

あらゆるものをプレイヤーに体験させるのがシミュレーションゲームの役割です。これは現実の世界にも影響を与えています。もともと飛行機操縦の訓練用であったシミュレータがゲームにされたり，またその逆の例もあります。バーチャルリアルティなどもこのシミュレーションが発展したものです。すでにあちこちで応用されていますが，この技術を使った新しい種類のアイドルすら生まれています。「現実をシミュレーションする」ということの延長線上には何があるのでしょうか。これこそシミュレートしてみたい題材です。

● シミュレーションゲームとは？

「現実に起きているあらゆるできごとをゲームとして体験できるようにする」これがシミュレーションゲームの本質です。もしあなたが学生なら，退屈な先生の授業を受け，遊びでストレスを発散させ，テストを受けたりすることが日常の生活だと思います。これをシミュレーションゲームにするのなら，「授業を受ける」「勉強をする」「遊ぶ」ことで自分の「能力」「評価」または「経験値」を高め，「テス

トを受ける」ことで具体的な「結果」が産み出され，そして「卒業」や「目的の学校に合格」したらそこでゲームの終了となるでしょう。これ以外にも「体験/経験」「成長」「仕事」など，何でもシュミレーションゲームにすることができます。

◆ライフゲームから進化

シミュレーションゲームの始祖ともいえるものが「ライフゲーム」です。1970年にMartin GardnerがJ.H.Conwayの「LifeGame」を『Scientific American』誌に紹介したのが始まりです。それ以後，多くのコンピュータで遊ばれています。このゲームの仕組みはとても簡単で，フィールドに生き物となるコマを置き，ほかの生き物が周囲にどれだけいるかで，その生き物の生死や繁殖するかどうかが決まるというものです。プレイヤーは始めに置く生き物の数や「周りにどれだけの生き物がいるとどういうことが起きる」という条件をプログラムとして設定します。この「影響し合うもの」と「条件となる値を与える」ことから「結果」を導き出すというルールは，いまある多くのシミュレーションゲームへ形を変えて受け継がれています。

◆現実のできごとを抽象化する

現在作られているほとんどのゲームは，必ずといっていいほど，どこかにシミュレーション的な要素を持っています。ロールプレイングゲームは「作られた世界を自由に旅する」シミュレーションといい換えることができます。まったく別の分野に見えるパズルゲームなども，現実のできごとを抽象化したシミュレーションゲームだといえます。「倉庫番」という有名なパズルゲームがありますが，これを例として考えてみればわかりやすいでしょう。私は車でスキーへいくとき，トランクに荷物を詰めるのにこれをいつも体験しています(笑)。

このようにシミュレーションというとあまりにもジャンルが広いのですが，ここでは「人を育てる」タイプの育成型シミュレーションゲームを題材にして，シミュレーションゲームで使われる「パラメータ」を考えてみたいと思います。

● シミュレーションゲームの中身

実際のゲームを見てみると，「パーツを置く」「1週間でやることを決める」といったプレイヤーが操作を行う「エディットモード」と，コンピュータが時間を自動的に進める「進行モード」というものがあるのがわかります(Fig. 2D-1)。これは

Fig. 2D-1 ●シミュレーションゲームの画面構成。ゲームプレイ時にはBGMと効果音が加わる

「ターン制」などともいわれます。これが明確に分かれていないゲームもありますが，その場合も基本的にはこのモードが1つにまとめられているだけです。

エディットモードでは，プレイヤーが自由に「値」を設定できます。街を作るゲームでは，マップ画面に好きなパーツを置くことで街を作っていきます。育成型なら教育したり働かせたりなどの「行動」を決めます。このパーツや行動などのプレイヤーが設定した情報は，プログラムとして見ればただの変数にすぎません。ロールプレイングゲームのキャラクタパラメータと同様に，対応する変数を各情報に対応する値に従って増減させているだけです。

時間が経つと画面上に表示されるキャラクタが成長した状態に変わることがありますが，これも変数を変えることで実現しています。育成型のゲームでは，この変数の値を直接グラフなどで見られるようになっていることがほとんどです。

◆キャラクタの持つパラメータ

プレイヤーが指示する「行動」によって「プログラム内にある変数の値に変化が加えられる」というように考えると，そこに「パラメータ」というものが見えてきます。

育成型のシミュレーションゲームをプレイしていると，「ある行動を取ると，環境が悪くなってキャラクタの成長が遅くなる」ということがあります。この現象に

何かの変数が関係しているとすれば，「環境へどの程度影響を与えるか」という変数が「行動」ごとにあり，キャラクタの持つ「環境状態」「成長度合」などの変数がこれによって左右されているはずです。つまり，このキャラクタが持つ「変数の固まり」こそが「パラメータ」なのです。

パラメータにはいろいろな種類があります。先ほど出てきたもの以外にも「服」「食べ物」といったイベント中に買うことができる「アイテム」，「行動」によってキャラクタに加えられる値もパラメータとして考えられます。

パラメータは，「何かされるたびに変化する可変的な値」，「何があっても変わらない固定的な値」の2つに分けることができます。成長速度など，行動の選択などによって影響が加えられるものは，その影響の度合いを値にして持っていないといけませんから「可変的な値」として扱います。逆にこれら可変的な値に影響を与える「このぐらい成長に影響を与える」などや，直接は可変的なパラメータとは関係ない「必要となるお金」といったものは固定的な値でもかまいません。

パラメータの変化は必ずしも「プラス」になるだけでなく「マイナス」にもなります。「行動」の種類によっては「こっちのパラメータをよくすると，別のパラメータが悪くなる」という具合に，相反するように設定されています。シミュレーションゲームの本質がここにあり，ゲーム作成のポイントにもなります。バランスよく「プラス」と「マイナス」を組み合わせることができれば，とてもおもしろいゲームになるでしょう。

◆ターンを繰り返してゲームを進める

プレイヤーがひととおり設定を終えると，時間を進めるようにゲームへ指示します。ゲームが進行モードになると，設定の内容に従って「アニメーション表示」やキャラクタのパラメータに変化を与えます。設定した行動がよければそのキャラクタは成長したりするでしょう。この処理をしている時間の単位を「ターン」といいます。時間の進行は，このターンを何度も連続させることで進められています。

ターン進行中に「病気になる」など「イベント」が起こることもあります。これらはキャラクタのパラメータや行動の内容などが引き金となって起きます。プレイヤーには操作することのできない裏パラメータで左右されることも多いようです。イベントが起きた結果もパラメータへと反映されます。

◆行動によって結果が変化していく

こうしてさまざまに手を加えた結果として，最終的に「エンディング」を迎え

ます。よい結果となるか悪い結果となるかはプレイヤーしだいです。これもパラメータの値に左右されますが，ゲームによってはイベントや選択した行動によってエンディングが変わることがあります。

シミュレーションゲームでは，パラメータがどのような変化をするかによって結果が変わります。この結果が予測できないところがシミュレーションゲームのおもしろさです。まったく同じ手順を踏めば同じ結果になりますが，長いターンの繰り返しになると，なかなかそうはいきません。パラメータに乱数を加えたりすれば，予測は不可能でしょう。また途中起きるハプニングも「ある手順によって」起きるというところまではわかりますが，それによって引き起こされる微妙な影響は全部わかるものではありません。これを知ろうとする謎解き要素もシミュレーションゲームは持っています。

シミュレーションゲームのデータ構造

1つの「キャラクタ」や「行動」には非常に多くのパラメータが関係します。1つの行動に対してそれぞれ変化させるパラメータの値を持たせるのはよいとしても，値が大量にあると管理がたいへんです。

そこでまず基本となる「キャラクタ」のパラメータを決めてしまいます。体力や魔力などといったロールプレイングゲームなどでもおなじみのものです。ゲームによってはここに「裏パラメータ」なるものがあります。これはプレイヤーには知ることができないパラメータで，おもに「戦闘時の勝敗」「エンディングの選択」などを決めていることが多いようです。ゲームの設定やプログラムに合わせて，この項目の変数を決めてください(List 2D-1，2D-2(P124))。

「行動」のほうは，その行動が実行された瞬間にキャラクタのパラメータが変わることになります。そこで「行動」と「それに対して増減させるパラメータの値」は，常に一組になるはずです。これをテーブル形式でまとめておきます(Fig. 2D-2)。「行動」を実行するときは，「行動処理用関数」にその行動と対応する「テーブルの番号」を渡して実行するようにします。アイテムも同じようにして管理します。

プレイヤーに選択された行動は，「月に4回」など1回のターンの中で複数回実行されます。これには「テーブルの番号」をそれぞれ決まった配列の「行動指定用テーブル」に入れるようにします(Fig. 2D-3)。設定したときに画面表示を変えたいときは，このテーブルを操作するための関数を用意しておき，そこで行うようにします。

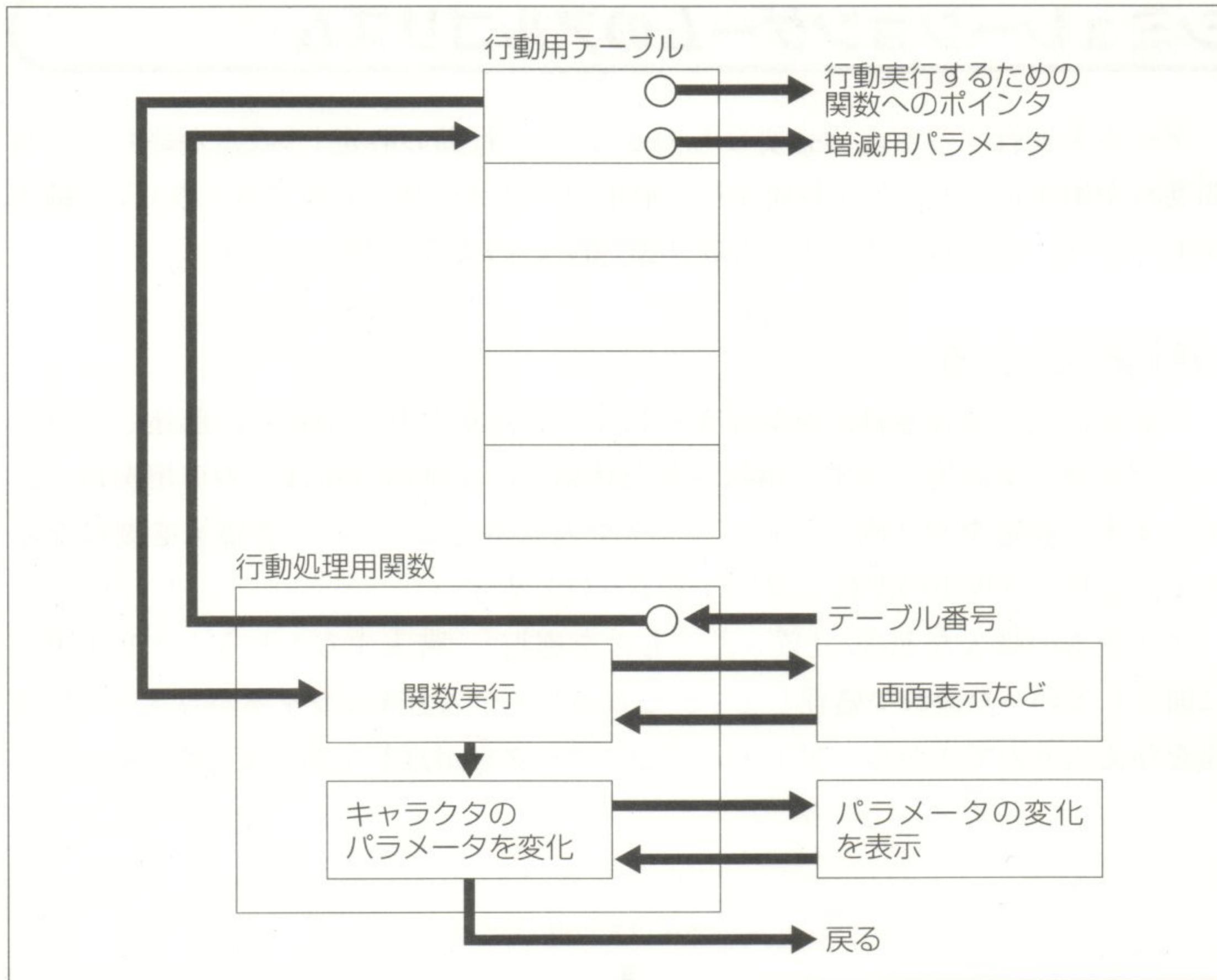



Fig. 2D-2 ●行動とそれによって増減する値をテーブルにし，行動を実行する関数にテーブル番号を渡して処理を行う

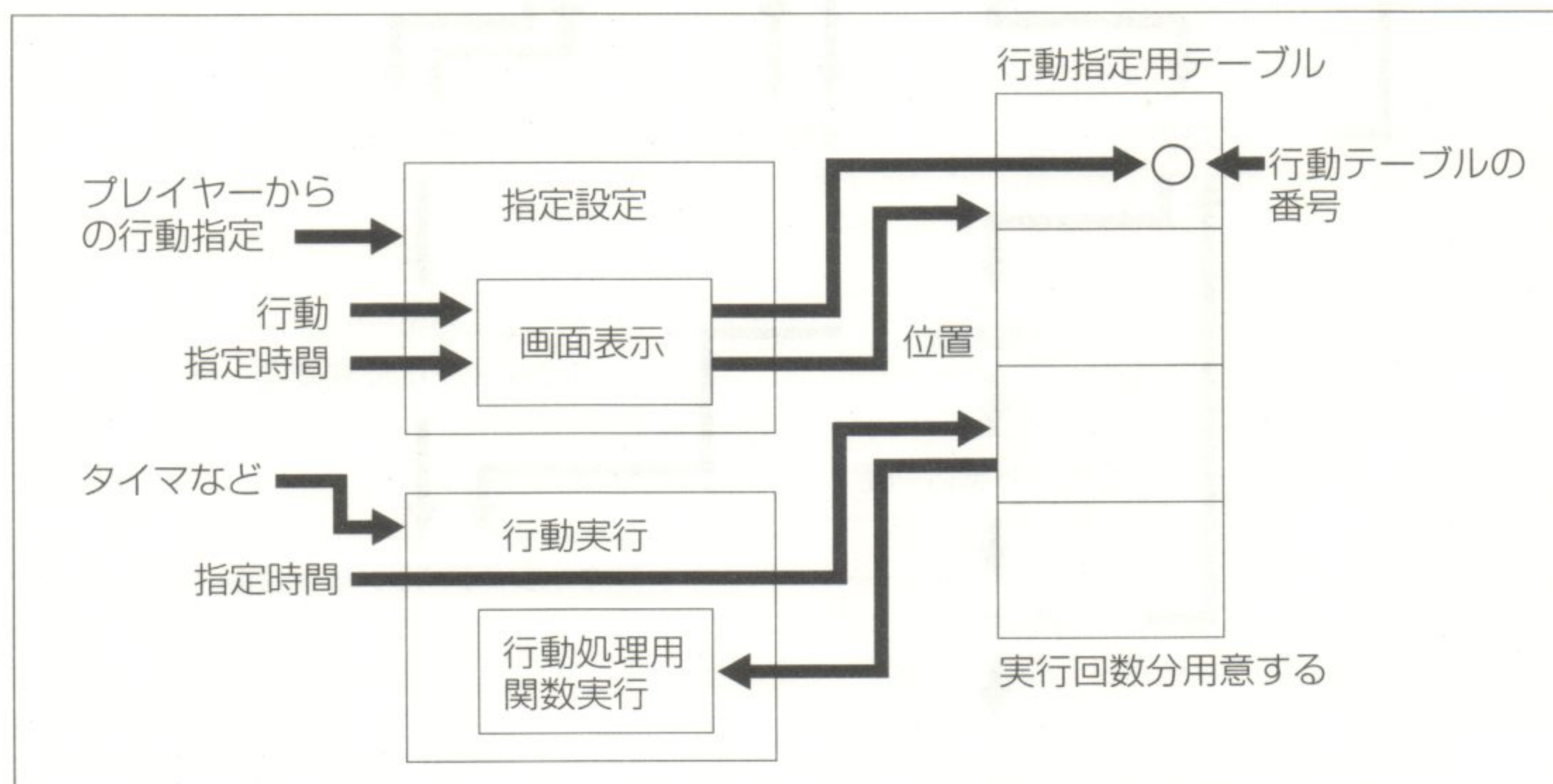



Fig. 2D-3 ● 1 回のターンで実行される回数分の行動指定用テーブルを用意し，そこに選択した行動などを収める。あとは行動用テーブルから対応するデータを引き出し，タイマに従って実行する

● シミュレーションゲームのアルゴリズム

ゲームを進行するうえで必要なものとして，「行動の設定」「設定が終わって時間進行が始まるとき」と「時間進行の最中でパラメータを計算するとき」の3種類があります。全体的なプログラムは(Fig. 2D-4)のように進みます。

◆行動を設定する

行動を設定するときは，プレイヤーによって選択された情報を，前述した専用の関数を使って設定します。選択された状態を示す画面表示はこの専用関数で設定します。設定をやり直したり，データを入れ換えるといった機能も必要になるので，それらも専用の関数で操作するようにします。

アイテムの購入などは，「買いたいものを選択」「購入するかどうかプレイヤーに聞く」といった流れで処理します。これらアイテムもキャラクタパラメータに変化を与えるものですから，アイテムごとにデータを用意しておいてください。

オープニング
行動選択状態
行動実行
行動選択
アイテム取得
行動結果
パラメータをプレイヤーパラメータに反映する
イベント実行判定
キャラクタパラメータ変更
終了判定
エンディング

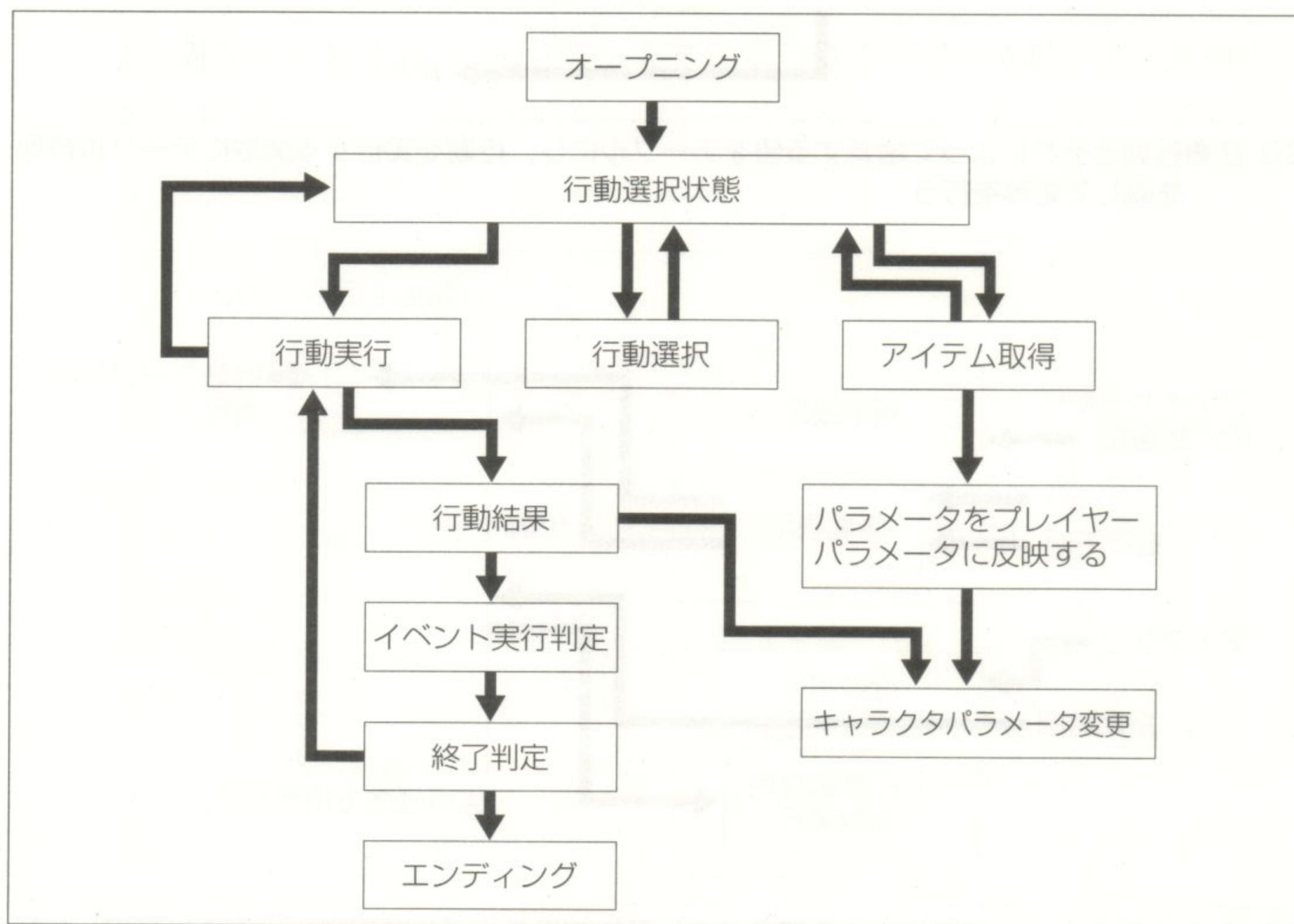



Fig. 2D-4●シミュレーションゲームのアルゴリズム

◆時間進行のターンでの処理

時間進行が始まるときは，設定されたテーブルに従って処理を動かすことになります。これはスクリプトにしなくてもかまいません。行動の設定状態によって「続いて同じ設定をされたら増えるパラメータを半減」といったことをしたい場合は，このときに判断するようにします。

各ターンの処理は，タイマなどによって少し間隔を置きながら各行動を実行します。アニメーション処理，ゲームを終えてもいいかどうかを調べる「終了判定」やさまざまなハプニングなどの処理も，このターンで実行します。これらは行動を実行するたびに，キャラクタパラメータなどをチェックするようにします。

◆パラメータを調整する

パラメータの個々の値を効率よく管理しながら調整を加えていくのには，「パラメータを操作するときは必ず専用の関数を使う」ことと「複数のパラメータを操作する関数は必ず1つにまとめる」ことが必要です。パラメータの操作では「あるパラメータが増減するとその影響を違うパラメータも受ける」ことが大部分を占めます。このため，何かのパラメータを操作するときは必ず関数を通すようにし，その関数の中でほかの影響を受けるパラメータもいっしょにまとめて操作します。

エンディングはパラメータが基になって決められます。たとえば「あるエンディングには，このパラメータがこの値になっていることが必要」など，その条件に当てはめるようにしてエンディングを選択します。またイベントによって「このエンディングになるようにしたい」ときは，それを裏パラメータとして持たせます。

● サンプルゲームの遊び方

天才的な頭脳を持つあなたは秘密結社「タクラム」に誘拐同然に連れ去られ，あることを依頼されます。人型アンドロイドをぜひ作ってほしい……。しぶしぶ同意したあなたは，アンドロイド作成のための情報を秘密結社に少しずつ売っていくかわりに，希望するパーツやお金/情報をもらいます。あなたの欲望どおりにアンドロイドを作成し，この腹立たしい悪の秘密結社を倒すことができるのでしょうか？

ゲームの世界観はこんなところです。起動すると，オープニングのあと椅子しかない部屋が画面に映ります。そのあと秘密結社からもらった仕度金を元にアンドロイドを作るためのパーツを買っていきます。ただし使えるお金が決まっているので，最初は最小限のパーツを買うようにします。また，どうしてもほしいものは秘

密結社のほうで非合法的に持ってきてくれることもあります。

途中あなたが籍を置く秘密結社の対抗組織が刺客を送ってきます。このときの戦闘はコンピュータが自動的に行います。敵の刺客の性能が，あなたの作っているアンドロイドよりも上回っていると負けてしまうので，適度に製作を進めつつできるだけ秘密結社とかかわらないようにしてください。エンディングは1種類しかないので，アンドロイドを作る過程と戦闘をゲームの中心にしてあります。

グラフィック表示部分にマウスカーソルを置き，クリックボタンを押し続けると，メッセージを早送りできます。

プログラムではたいしたことはしていません。画面の表示にはスプライトを利用しています。これ以外に，パラメータ関係の処理と時間進行，そのほか戦闘シーンなどを加えました。内容として，そのほとんどがパラメータ処理のルーチンになっています。単純な作りなので，これを参考にしてもっと複雑なゲームを作るのもおもしろいでしょう。

List 2D-1 ●キャラクタ用のデータ構造体（Delphi）

```
{ 型宣言 }
type
  TCharacterParam = record              { パラメータの定義          }
     HP: integer;                       { 体力                      }
      HPmax: integer;                   { 体力最大                  }
      Weariness: integer;               { 疲労                      }
      Wearinessmax: integer;            { 疲労最大                  }
      Wisdom: integer;                  { 知力                      }
      Wisdommax: integer;               { 知力最大                  }
      Shield: integer;                  { 防御力                    }
      Shieldmax: integer;               { 防御力最大                }
      Attack: integer;                  { 攻撃力                    }
      Attackmax: integer;               { 攻撃力最大                }
      Quick: integer;                   { 俊敏                      }
      Quickmax: integer;                { 俊敏最大                  }
      Glamor: integer;                  { 色気                      }
      Glamormax: integer;               { 色気最大                  }
      // これ以降は裏パラメータ
      Lucky: integer;                   { 運                        }
      Luckymax: integer;                { 運最大                    }
      Batl: integer;                    { 戦闘頻度                  }
      Batlmax: integer;                 { 戦闘頻度最大              }
   end;

   TCharacter = record                  { キャラクタの定義          }
      gra : array[0..CHAR_TBLMAX] of ^byte;
                                        { グラフィックデータ        }
```

▼

```
        Name: String;           { キャラクタ名              }
        Param: TCharacterParam; { パラメータ                }
    end;

    TExecItem = record          { 行動の定義                }
        gra : array[0..EXEC_TBLMAX] of ^byte;
                                { グラフィックデータ        }
        Name: String;           { キャラクタ名              }
        pfunc: Tfunc;           { 行動実行用の関数ポインタ  }
        Param: TCharacterParam; { パラメータ                }
    end;
```

List 2D-2●キャラクタ用のデータ構造体（C/C++）

```
/* 型宣言 */
typedef struct {                        /* パラメータの定義        */
    int HP;                             /* 体力                    */
    int HPmax;                          /* 体力最大                */
    int Weariness;                      /* 疲労                    */
    int Wearinessmax;                   /* 疲労最大                */
    int Wisdom;                         /* 知力                    */
    int Wisdommax;                      /* 知力最大                */
    int Shield;                         /* 防御力                  */
    int Shieldmax;                      /* 防御力最大              */
    int Attack;                         /* 攻撃力                  */
    int Attackmax;                      /* 攻撃力最大              */
    int Quick;                          /* 俊敏                    */
    int Quickmax;                       /* 俊敏最大                */
    int Glamor;                         /* 色気                    */
    int Glamormax;                      /* 色気最大                */
    // これ以降は裏パラメータ
    int Lucky;                          /* 運                      */
    int Luckymax;                       /* 運最大                  */
    int Batl;                           /* 戦闘頻度                */
    int Batlmax;                        /* 戦闘頻度最大            */
} TCharacterParam;

typedef struct {                        /* キャラクタの定義        */
    char *gra[CHAR_TBLMAX + 1];         /* グラフィックデータ      */
    char Name[255];                     /* キャラクタ名            */
    TCharacterParam Param;              /* パラメータ              */
} TCharacter;

typedef struct {                        /* 行動の定義              */
    char *gra[EXEC_TBLMAX + 1];         /* グラフィックデータ      */
    Tfunc pfunc;                        /* 行動実行用の関数ポインタ */
    TCharacterParam Param;              /* パラメータ              */
} TExecItem;
```

Section 5 シューティングゲーム

シューティングゲームでは極限まで速さが求められます。では，なぜシューティングゲームでは速さが求められるのでしょうか？　なぜ作るのが難しいといわれるのでしょうか？　シューティングゲームのアルゴリズムとデータ構造を深く探ってみしょう

撃って，かわして，倒す

「ものを何かで撃つ」ということに人は快楽を覚えます。さらに「敵を追い詰める」「敵の攻撃をかわす」といったスリリングさもたまらない快感でしょう。これはどんなに理性で隠蔽しようとしても，人間なら必ず持つ不変的なもののようです。大昔，人類が動物を狩りでしとめていたころの名残かもしれません。もちろん現代では，対象が何であれ，こういうことをしてはいけないのですが，私が尊敬する偉大なコメディアンの言葉を借りるなら「わかっちゃいるけどやめられない」というところなのでしょう。

この本能的な衝動を巧みにくすぐるシューティングゲームについて，詳しく取りあげます。

シューティングゲームは速さを極限まで求めます。もしマシンの速さのほうがゲームを上回ったら，さらに凝った処理やより深い内容をゲームへ加えていきます。加えられた部分が多ければ多いほど，CPUやビデオボードの処理が追い付かなくなります。そしてまたマシンが速くなり……と，こうしたイタチごっこはすでに何年も続けられているのです。プレイヤーは純粋にゲームを楽しむことはできても，プログラマはよりよいゲームを作るために，これからもいばらの道が続くことでしょう。プログラマは決して気楽な商売じゃありません。

シューティングゲームとは？

「プレイヤーの操縦する自機が空間を移動しながら，つぎつぎに現れる敵に向かって弾を撃ちまくる」これがシューティングゲームです。「キャラクタとなるもの（自機）または銃の照準などをプレイヤーが操作」「プレイヤーは敵を倒す弾を撃つ

ことができる」「敵は大量に出てくる」「敵も弾を撃ってくるのでそれを避けなければならない」というポイントからゲームが成り立っています。

シューティングゲームのおもしろさは，「敵の弾がいつ当たるわからない」といった日常では味わえない緊迫感の中で「敵の攻撃を避けつつ，弾を撃ちまくる」というところにあります。特定の敵とだけ対峙するゲームもありますが，シューティングゲームでは，とにかく大量の敵が出てきます。しかもこれがリアルタイムで反撃してくるのです。この「ひきゃ〜〜っ」という何ともいい表すことができない追い詰められた感じの中で，多くの敵をばったばったと撃ち落としていき，そして最後の敵ボスを苦労して倒したときには，爽快感とともに勝利の快感を得ることができるでしょう。またシューティングゲームは中毒性が強く，慣れたプレイヤーはより難しいゲームを求める傾向があります。

◆シューティングゲームの分類

シューティングゲームのおおまかな分類方法として，背景の移動方向から「縦スクロール」「横スクロール」または「全方向スクロール」などがあります。最近では，画面の表示を現実の人間の視点とまったく同じにして，背景が多重スクロールしたりズームしたりするゲームが多く見られます。一部のゲームでは，映画などで使われるカメラアングルを参考にしているものもあるようです。

自機や敵の種類は飛行機といった機械/乗り物から，人，怪物などさまざまで，そのゲームのストーリや設定に左右されます。ストーリはロールプレイングゲームのように凝ったものから，シューティングゲームの王道ともいえるスペースオペラ風なものまで，非常に種類が多いのも特徴です。最近は傾向としてSF調よりもファンタジー調の設定が多く見られます。

自機が弾を撃つときは「弾を撃つ方向は固定」「弾を撃つ方向はユーザが操作する」という2つの方法に分かれます。多くの場合，方向を指示する十字キーは1組しか用意されていないので，どちらかの目的にしか使えないためです。この選択はゲームシステムや設定などにより決められます。

● シューティングゲームの中身

シューティングゲームでの処理を実際のゲームを見ながら調べてみましょう(Fig. 2E-1)。

敵キャラクタ
弾
戦え!僕らの看護婦さん!!
ゲーム(G) 表示(V) ヘルプ(H)
Score
83
得点表示
スクロール方向
E-Level
A-Level
背景
自機
(プレイヤーが操縦)
●BGM
●効果音
パラメータ
(残り自機数など)

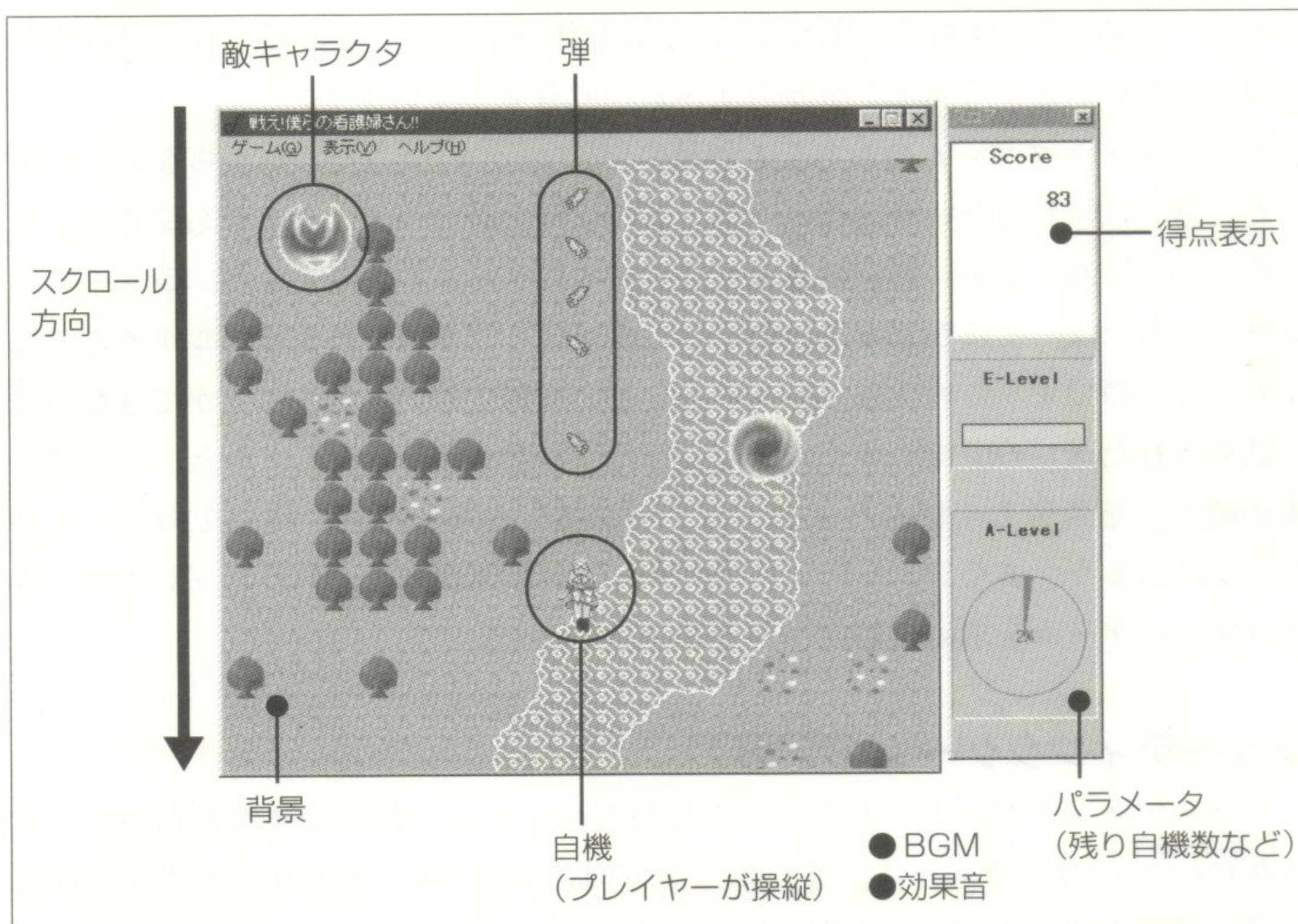



Fig. 2E-1●シューティングゲームの画面構成。ゲームプレイ時にはBGMと効果音が加わる

画面にはまず敵と自機がいて，それとは無関係にスクロールしている背景があります。それに，得点の表示や自機の残り数の表示などが加わります。シューティングゲームでは，ロールプレイングゲームと同じく小さなパーツを並べることにより，背景やキャラクタを画面上に描画しています。ただしロールプレイングゲームとは違い，シューティングゲームのキャラクタ移動はパーツの大きさに縛られません。ほとんどのパーツはある決まったドット(ピクセル)数ごとに移動しています。このドット数を「移動量」といいます。

背景のスクロールもパーツごとではなく，なめらかにスクロールしています。パーツを組み立てるところはロールプレイングゲームと同じですが，画面へ描画するときは，パーツを一気に描画するのでなく，あるライン数ごとに分けて少しずつ転送します。転送するライン数が小さいとスクロールの速度が遅く，多いと速く流れているように感じさせることができます。

◆スプライト機能

近ごろでは複雑な背景や何重にもキャラクタを合成してできているきれいな画面のゲームが増えています。このきれいな画面を実現させている機能のひとつが「ス

プライト」です。

スプライトは「透明な小さなパーツの上にキャラクタの絵を描き，それを画面の上に何枚か組み合わせて並べる」という仕組みになっています。1枚の大きな透明のフィルムに全部の絵を描くのではなく，フィルムを小さなパーツに分割して，そのパーツ1つひとつに絵を描いていく感じです。1つのパーツだけ使えば弾やキャラクタなど小さなもの，複数組み合わせて使えば背景など大きなものを表示することができます。

スプライト機能は，一部のパソコンやコンシューママシン，アーケードマシンには，ハードウェアの機能として持たせてあります。WonderSwanのような携帯ゲーム機にも搭載されています。CPUに負荷をかけないため，より速く合成処理をすることが可能です。Table 2E-1に具体的なスプライトのスペック例を示しました。なお，DirectXでは少し形は違いますが，スプライト的な機能を利用することができます。

Table 2E-1 ●ハードウェアスプライトの例

Sony Play Station	
グラフィックサイズ	1×1～256×256ドット
色数	4/8ビットCLUT/15ビット・ダイレクト/32,767色 (ヒデオ機能そのものの最大発色数は1677万色)
描画性能	4000個/画面（8×8ドットのスプライトを1/60秒で表示）
特殊効果	回転，拡大縮小，歪み，透明度，フェードイン/フェードアウト，優先順位，縦/横スクロール
フレーム/ライン最大数	制限なし
パーツ定義最大数	制限なし

注) CLUT - Color Look-Up Tables　16/256色カラーパレット

◆当たり判定

敵はプレイヤーが操作する自機に対して，いろいろな攻撃を仕掛けてきます。もし敵が出す弾や敵にぶつかると，自機は壊れたり爆発したりします。この「弾や敵、障害物」などに当たったかどうかを判断するのが「当たり判定」です。

判定の方法にはいくつか種類があります。単純に各キャラクタを動かしたときにそこに何があったのかを見るものでも，仮想画面を用意したり，専用のソートされたテーブルを持たせたりする方法などがあります。変わったところでは，キャラクタ同士の色の重なり具合から判定する方法も使われているようです。

◆アイテムでパワーアップする

プレイ中に自機が何かのキャラクタを取るとパワーアップができるゲームがあります。このキャラクタを「アイテム」といいます。自機には「残存する自機の数」「自機の弾の種類」などといったものが変数(パラメータ)として用意されています。プログラムでは，自機がアイテムを取るとアイテムの種類に応じてパラメータを変化させます。このあたりはシミュレーションゲームやロールプレイングゲームでのパラメータ管理と同じです。自機がアイテムを取得したかどうかを調べるには，弾の当たり判定と同じ方法が取られています。

ゲームに登場するキャラクタはほかにもあります。あるタイミングでないと通ることができない歯車や，壊さないと進めない壁といった「仕掛け/罠(トラップ)」です。こうしたものは難易度を上げたり，隠しアイテムを入れたりするのにも使われています。表示などの処理は敵キャラクタと同じです。

◆スクリプト

敵が出現するタイミングをよく見ると，ランダムに出現するというよりも，ある場面ごとに出てくる敵の種類が決まっていることがわかります。この「ある場面になったらどこに特定の敵が出現する」といった情報は，まとまった1つのデータとして置かれています。このデータのことを「スクリプト」などと呼びます。

スクリプトの中身は単純です。たとえば配列の1行に「敵の種類」「敵の出現位置」「進行方向」などといったデータをまとめて記述し，これを何行か重ねてやります。タイマによりある一定時間ごとにこの配列を1行ずつ読み取り，そのデータに従って敵を出す処理をします。読み取りが終わったあとは配列のポインタを1つ先へ進め，次にタイマが起動したときのために備えます。これをゲームが終わったり場面が変わるまで繰り返しています。

◆スピードが最重要ポイント

シューティングゲームを作るうえでもっとも要求されるのが「スピード」です。キャラクタ移動などのゲームに必要な各作業は，それぞれ単体で行えばとても速く処理することができます。しかしゲームでは，各作業はまとめて行わなければなりません。1つの作業が短時間で終わっても，それが何回も繰り返されるのですから，それが積み重なって結果的には長い時間が必要となります。

処理の遅さはシューティングゲームの命である臨場感を奪います。ここにシューティングゲーム作りの難しさがあるわけです。違うジャンルのゲームでも処理の速

さは求められますが，シューティングゲームではスピードが目に見えてしまうため，ことさら厳しいものになります。

スプライトのデータ構造

ここではスプライト機能がないマシンのために，ソフトウェアだけで実現できる「擬似的なスプライト」を作ります。ここから逆に，ハードウェアが持っているスプライトの仕組みを考えてみるのもおもしろいと思います。

◆スプライトに必要な設定

ハードウェアが持っているスプライトの処理は，「合成するグラフィック」「座標指定」などのデータを「レジスタ」で指定することから始まります。レジスタはある決まった数だけ並んだテーブルの形になっています(Fig. 2E-2)。グラフィックのサイズは8×8を基本として，それを何倍かしたものがポピュラーです。座標はグラフィックを表示したい位置の座標を指定します。この座標を少しずつ変えていくことでキャラクタが動いているように見せることができます。

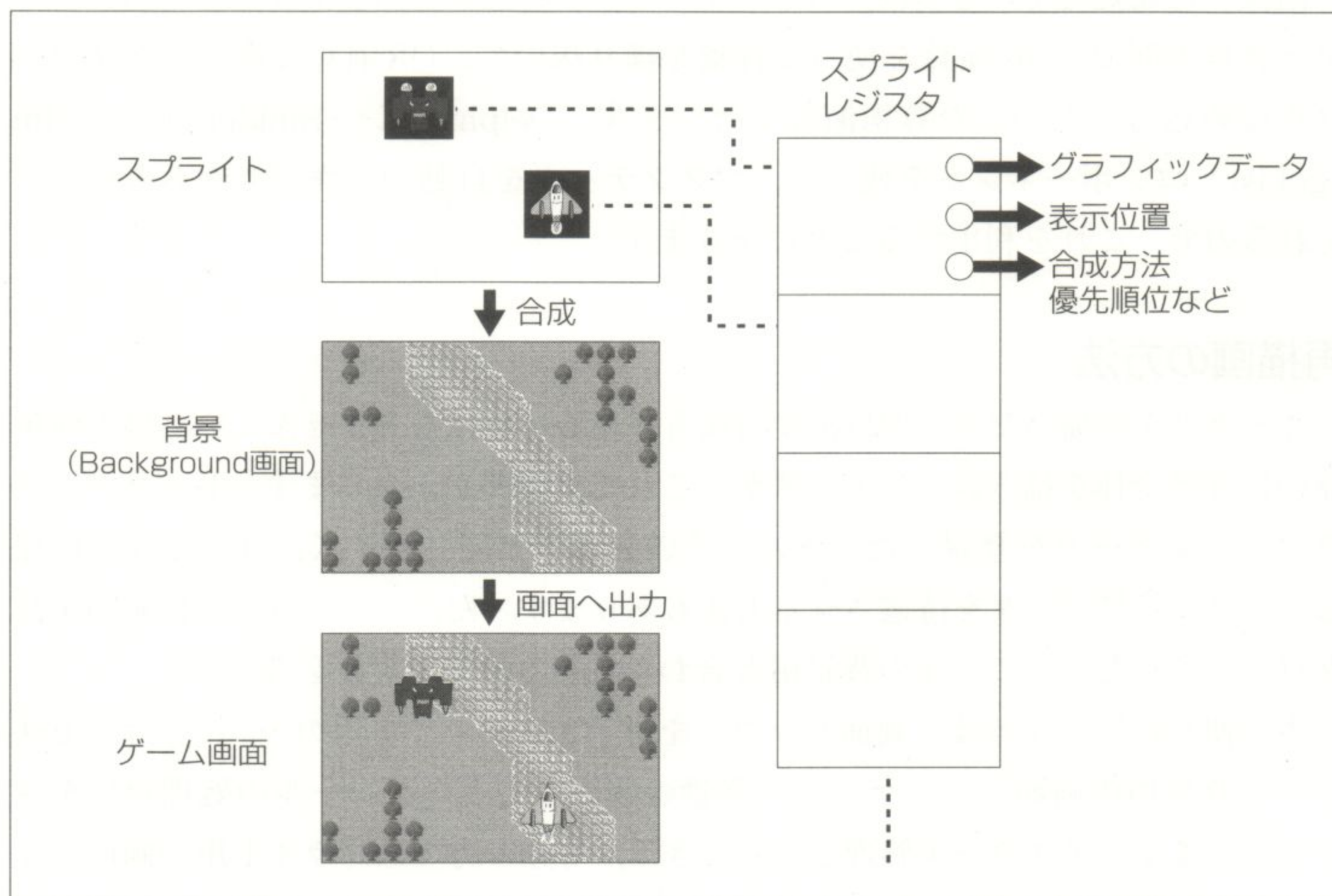



Fig. 2E-2 ●スプライト処理の仕組み。スプライトレジスタの情報を基に，表示するグラフィックや座標，そのほかのデータを指定して各パーツを背景と合成する

合成段階でグラフィックデータを拡大縮小や回転させる機能を持つスプライトでは，その方法をほかのデータと同じように設定します。また「優先順位」というものもあります。スプライトに指定されたグラフィックデータがいくつか重なるときは，この順位が優位にあるものが上になるように合成されます。これらのレジスタに値を設定すると，あとはスプライトを管理する側で合成処理を行います。プログラム側は合成作業に関しては何もしなくてかまいません。

◆擬似的にスプライトを作る

以上の仕組みをプログラムで実現させてみます。まずデータを指定するためのレジスタが必要です。これは1つの構造体として定義しておきます。さらにこの構造体をいくつかまとめてテーブルにして使います。背景となるデータはRPGと同じように画面バッファへ組み立てる方式を取るので，画面バッファをポインタとして登録することにします。そうしてできあがったのがList 2E-1，2E-2(P138)です。

● スプライトのアルゴリズム

画像の合成処理そのものは，ロールプレイングゲームと同じように「マスクデータと背景の間でAND演算を使って背景を繰り抜いて，OR演算でキャラクタデータをはめ込む」といった算術演算を行います。DelphiやC++Builderでは「TImageList」コンポーネントを使うと，マスクデータを自動的に作ってから合成してくれるので，これを利用することになります。

◆再描画の方法

キャラクタを描いたら，次は消す方法も考えなければなりません。いちばん簡単なのは背景全体を描き直すことですが，これでは時間がかかります。レジスタへグラフィックデータを登録したときに，その大きさと同じサイズだけメモリを確保し，そこへ背景データを待避させる方法もあります。ただしこの方法では画面をスクロールさせたときに，元の背景用と合わなくなるので工夫が必要です。

よく使われているのは，画面バッファを背景とスプライト用の2つに分ける方法です。背景用の画面バッファでは，背景の組み立てとスクロールの処理だけをさせておきます。スクロール処理をするときは，もう一方のスプライト用の画面バッファにも同じ処理をします。キャラクタを消すには，背景用の画面バッファからキャラクタと重なる部分だけスプライト用画面バッファへコピーします。2つのバ

ッファで背景スクロールの同期を取る必要がありますが，こうすると何も考えずいつでもキャラクタを消すことができるので便利な方法です。さらに発展させて，このバッファの役割を交互に切り換えて画面出力に使うこともあります。

◆扱うパーツ数の制限

ソフトウェアで実現したスプライトは画面バッファ上で行われるので，ハードウェアで行われているスプライトとは違い，処理できるパーツ数などに制限はありません。しかし，いくつかの処理にはCPUパワーを使います。そのため「実用的な速度」を維持するには，使うことができるパーツ数などにやはり限界ができてしまいます。

● スクリプトのデータ構造

シューティングゲームで使うスクリプトは2種類あります(Fig. 2E-3, 2E-4)。「敵の出現位置」を管理するものと，「敵の移動方法を決める」ものです。敵は種類によっては，ぐるっと回転したり，そのまま自機へ突っ込んできたり，ある場所にずっと止まっていたりします。こうした「敵の性格」ともいえる「移動方法」をスクリプトデータとして用意しておきます。

タイマ

移動量	移動方法	カウンタ
X: 1, Y: 1	ノーマル	0.

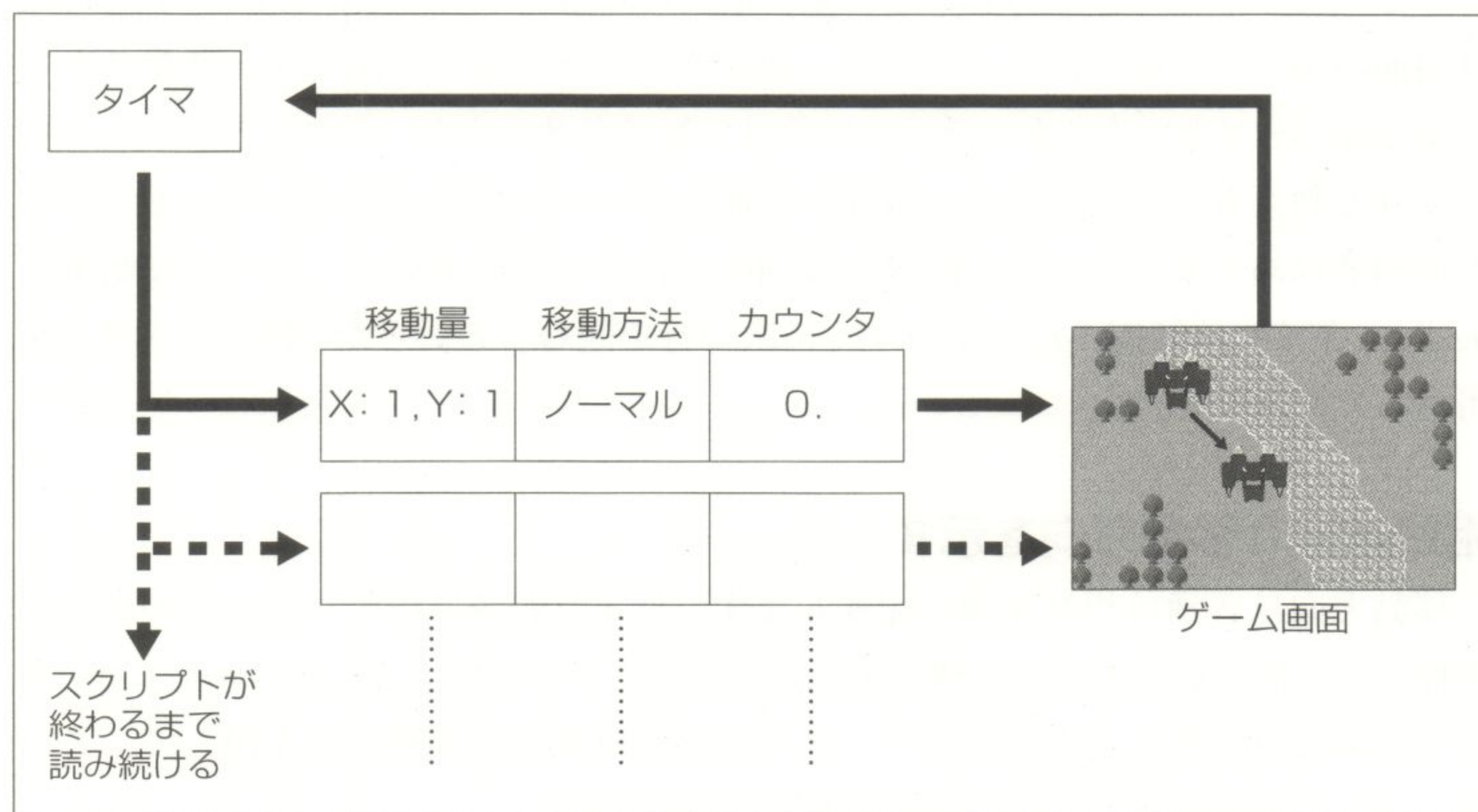
ゲーム画面

スクリプトが
終わるまで
読み続ける

Fig. 2E-3●キャラクタの移動用のスクリプト。キャラクタの移動量などを収めたスクリプトデータを用意する。それをタイマを使って一定間隔ごとに1ラインずつ読み出し，読み出したデータに従って画面表示の処理を実行する

タイマ

出現座標　キャラクタ　次のデータをすぐに読むか？

X:0,Y:0　キャラクタA　False

ゲーム画面

スクリプトが終わるまで読み続ける

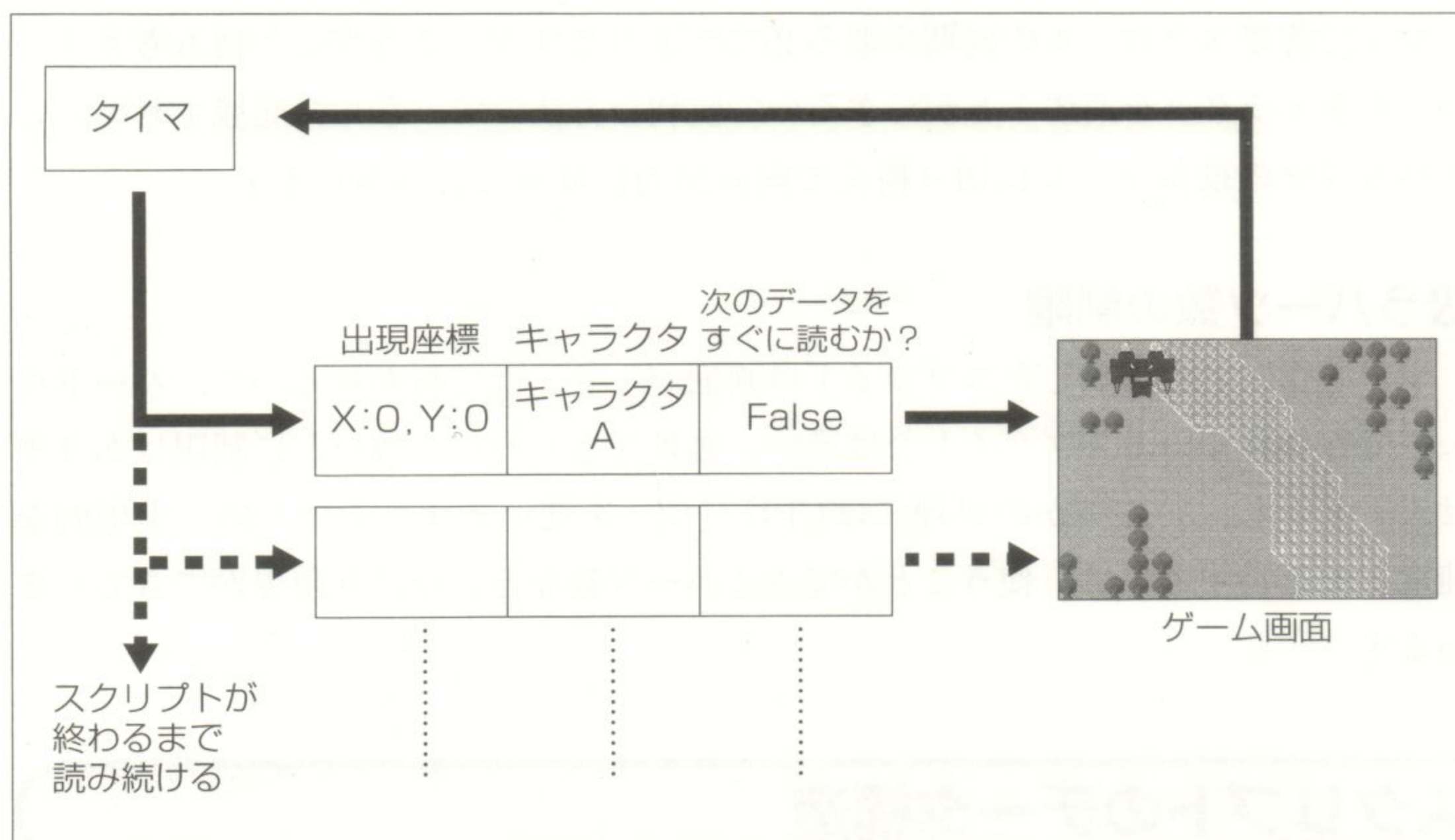



Fig. 2E-4●キャラクタの出現用のスクリプト。出現座標，出現するキャラクタなどを収めたスクリプトデータをタイマを使って処理を行う。ゲームでは複数の敵を出現させる必要があるため，次のデータを即座に読み出すかどうかのデータもスクリプトに収める

始めはタイマによって参照される1回当たりのデータ量から定義します。「敵の種類」はあらかじめ1つのテーブルにまとめておくので，スクリプトに記述する「敵の種類」はそのテーブルの引数になります。ぐるぐる回るのが「0」，突っ込んでくるのが「1」というようにです。敵が最初に画面上へ出てくる位置「出現位置」は画面の端を示す座標にしておくと，敵がやってきた感じが出ます。背景スクロールではパーツをいくつかのラインに分けて転送させますが，敵キャラはとくにゆっくりと動くもの以外なら，いきなり画面上にパーツ全部を出現させてもそれほど違和感はありません。また敵は複数同時に出すことも考えて「タイマに制御を戻す」「タイマに制御を戻さず，すぐに次の行を読む」といった2種類のフラグを用意しておきます。

◆相対座標で移動方向を示す

移動方向は普通は相対座標で記述します。タイマが読み出すたびに敵の現在の座標へこの値を加えます。自機に突っ込んでくるものではここにはタイマ1回で移動する量だけ記述します。そして，ゲーム中で自機との座標の差を出して，この値よりも上になるときだけ対応する座標に加えます。

こうしたデータを構造体にまとめて，テーブルの状態にして使います。移動方法を記述したスクリプトは敵の種類ごとに用意しておきます(List 2E-3, 2E-4(P139))。

● キャラクタのデータ構造

画面中に多数出てくるキャラクタは，それぞれ必要なデータを一括して管理する必要があります。データを参照する際に「アニメーションパターン」「敵や弾の移動方法を示したスクリプトデータ」「攻撃方法」「敵を撃ち落としたときの得点」といったものがキャラクタごとにまとまっていると便利です。これらはそれぞれ「キャラクタの種類を引数にしたテーブル」にまとめておきます。アニメーションパターンの格納方法はRPGと同じです。

一方で，出現した敵キャラクタや弾のすべてに個別のデータを持たせたいことがあります。とくに移動用のテーブルを参照するポインタは必要です。これもテーブルにしておきます。画面上に出すことができるキャラクタの数はかぎられるので，固定長の配列でかまいません(Fig. 2E-5，List 2E-5，2E-6(P143))。このテーブルはもう1つ用意して弾の管理にも使います。

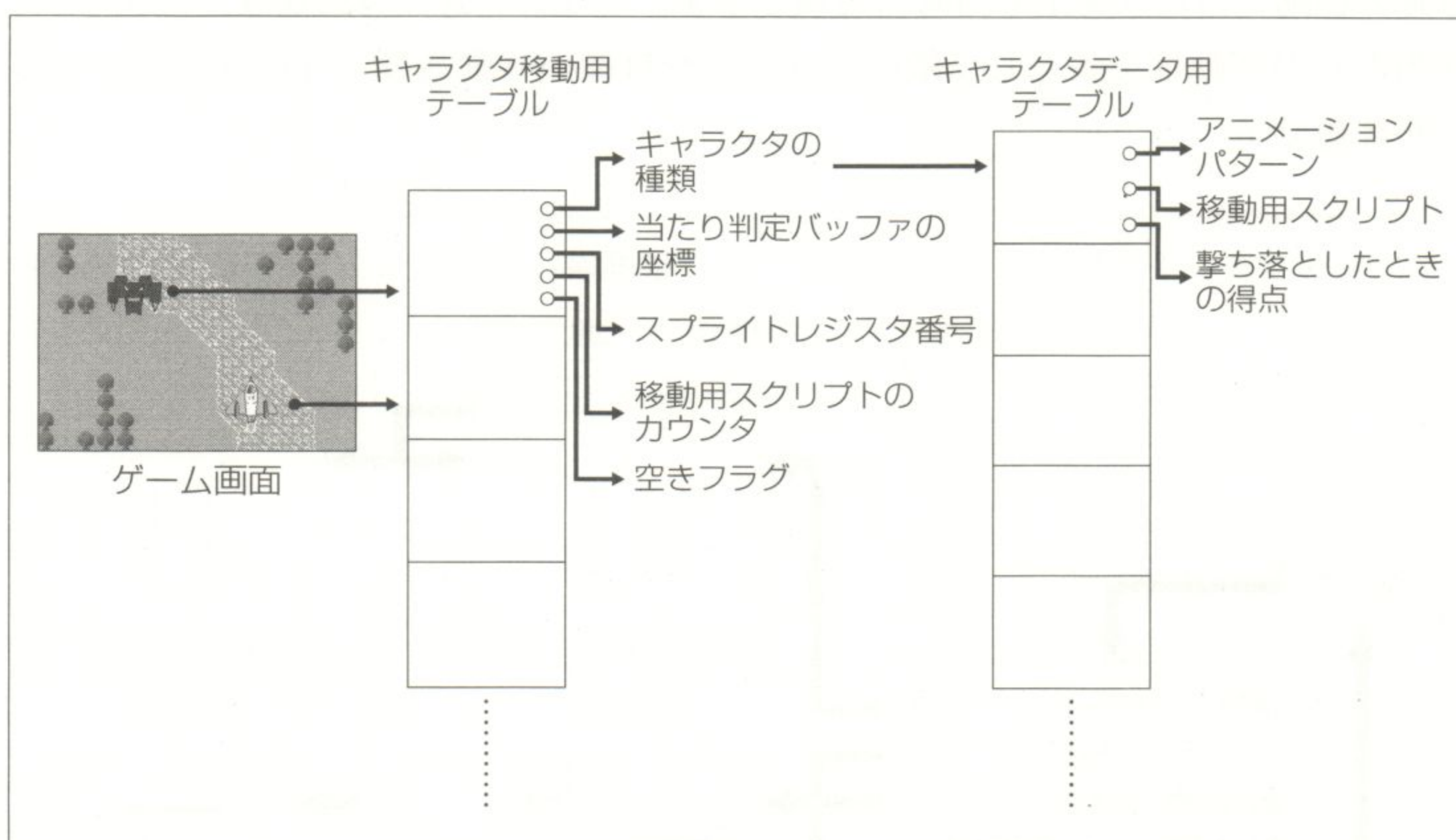



Fig. 2E-5 ●キャラクタごとに必要なデータをまとめたテーブルを用意し，画面に表示することができるキャラクタの数だけ用意した移動用のテーブルと合わせて処理を行う

◆データを消すための処理

「敵キャラクタが倒された」「弾が画面外に飛んでいった」などデータを消すときは，そこにはもうデータが入ってないことを示す「空きフラグ」を立てておきます。新しくキャラクタを出すときは，このフラグが立っているところに優先してデ

ータを入れるようにします。もし敵キャラクタが弾に当たったときは，爆発したシーンなどを表示するため，「爆発シーンのキャラクタ」に切り換えて，そのあとでしばらくしてから空きフラグを立てることになります。

● シューティングゲームのアルゴリズム

ここまで出てきたデータ構造を処理するアルゴリズムをまとめてみましょう(Fig. 2E-6，List 2E-7，2E-8(P144))。

背景の処理はロールプレイングゲームのときと同じですが，さらに縦と横のパーツを1ライン分並べたバッファを用意します。スクロールが進んで次のパーツが必要になったとき，まずこのバッファへパーツを1ライン分組み立ててから少しずつ転送していきます。この処理が終わったらスプライトへ渡します。

タイマは背景スクロールと敵キャラクタの出現/移動用に使います。このほかにも弾の移動と当たり判定も同時に処理します。ちなみに弾の移動速度は，タイマ1回当たりの移動量を多くして敵キャラクタや自機の移動速度よりも速くしているのが普通です。

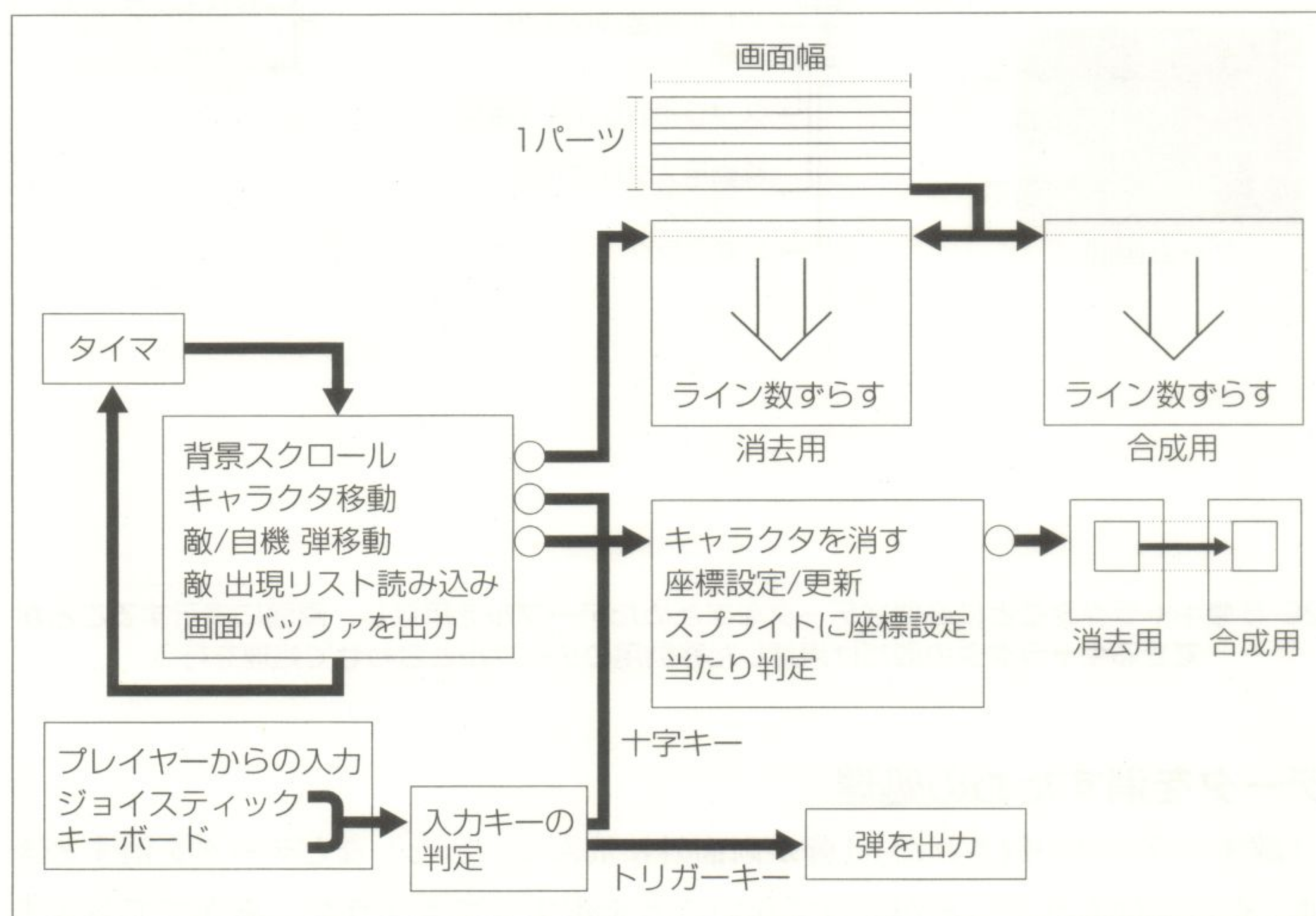



Fig. 2E-6 ●シューティングゲームのアルゴリズム

◆自機のキー操作

自機はキー操作により「弾の発射」「自機の移動」を行います。移動はキーデータによりスプライト上の座標を増減させます。このときに敵や障害物に当たったかどうかの判定もいっしょに行います。弾はトリガーキーが押されたときに弾移動用テーブルとスプライトへデータを設定します。

なお，この処理をする前にキーボードやジョイスティックの入力はすべてまとめておいて，必要なら別に定義した内部のキーコードに変えておきます。

◆当たり判定の方法

弾の当たり判定は前述のようにいろいろ方法はあるのですが，ようするに「キャラクタ同士の座標が重なるかどうか」ということを調べているだけにすぎません。当たり判定にいろいろな方法があるのは，より速く判定できるようにとプログラマがさまざまな工夫を凝らした結果です。

もっとも簡単な方法では，敵キャラクタを移動させたあと，弾を移動するときに，「敵の座標」「これから移動する先の弾の座標」「敵の座標＋敵のサイズ－弾のサイズ」を縦横の座標で調べます。この方法では，弾1つに対し敵の数だけ検索しなければなりません。検索する手間を省くために「画面を移動量ごとに区切った2次元配列にして，キャラクタの座標に対応した地点にフラグを置き，弾の進行方向の座標からこのフラグを取れば敵がいるのかどうかわかるようにする」方法が使われています。または数学の基礎的知識が必要になりますが，物体を「円」や「線」として見立て，それが近接しているかどうかを計算で求める方法もあります。いずれにしろめんどうなことには違いありません。どの方法でも最適化が必要ですし，ゲームの内容によってケースバイケースで仕組みを変えなければならないからです。ハードウェアやライブラリで当たり判定を提供しているのなら，そちらを使ったほうが簡単です。

◆いかに速く処理するかが重要

Fig. 2E-6を見れば，ほかのゲームと比べても工程が多いのがわかるはずです。とくにタイマから呼び出される処理は，できるだけ短い時間内に済ませる必要があります。どの処理も省けないので，処理ごとにわずかでも速くなるようにしなければなりません。

ここで紹介したさまざまな方法は「いかにして速くするか」を目的として作られたものばかりです。ゲームにするときは，この方法に加えて，より速くするため

ハードウェアの機能を利用して最速になるようにしています。

● サンプルゲームの遊び方

縦スクロールタイプのシューティングゲームにしてみました。1面しかありませんが，ルールなどはだいたい普通のゲームと同じです。

画面処理にはDirectXを使わず，DelphiとC++Builderのグラフィックコンポーネントを利用しているだけです。Pentium/133MHz + S3 ViRGE/4Mといった環境でそれなりに動くようにしていますが，これよりも遅いマシンだとゲームにならないかもしれません。なおスピード調整はメニューバーかパラメータ表示ウィンドウのボタンから「設定」を選んでください。

List 2E-1 ●擬似的なスプライト（Delphi）

```
{ 定数宣言 }
const
    SPREGTBLMAX = 128;

{ 型宣言 }
type
    TSpReg = record                 { TSpReg 構造体の定義 }
        x: integer;                 { 画面上の横位置     }
        y: integer;                 { 画面上の縦位置     }
        Width: integer;             { 横サイズ           }
        Height: integer;            { 縦サイズ           }
        gra: ^byte;                 { グラフィックデータ }
        mode: integer;              { 合成方法           }
        priority: integer;          { 優先順位           }
        FlagEmpty: boolean;         { 空いているかどうか }
        tag: integer;               { プログラマで利用可能 }
    end;

{ 変数宣言 }
var
    { スプライト用レジスタのテーブル }
    SpRegTbl: array[0..SPREGTBLMAX] of TSpReg;

    { スプライトで使用する画面バッファ }
    Screen: ^byte;
```

List 2E-2 ●擬似的なスプライト(C/C++)

```
/＊ 定数宣言 ＊/
#define SPREGTBLMAX  128


/＊ 型宣言 ＊/
typedef struct {               /＊ TSpReg 構造体の定義   ＊/
    int x;                     /＊ 画面上の横位置       ＊/
    int y;                     /＊ 画面上の縦位置       ＊/
    int Width;                 /＊ 横サイズ             ＊/
    int Height;                /＊ 縦サイズ             ＊/
    unsigned char ＊gra;       /＊ グラフィックデータ   ＊/
    int mode;                  /＊ 合成方法             ＊/
    int priority;              /＊ 優先順位             ＊/
    int FlagEmpty;             /＊ 空いているかどうか   ＊/
    int tag;                   /＊ プログラマで利用可能 ＊/
} TSpReg;

/＊ 変数宣言 ＊/
/＊ スプライト用レジスタのテーブル ＊/
TSpReg SpRegTbl[SPREGTBLMAX + 1];

/＊ スプライトで使用する画面バッファ ＊/
unsigned char ＊Screen;
```

List 2E-3 ●スクリプトのデータ構造(Delphi)

```
{ スクリプトのデータ構造の例 (Delphi/ObjectPascal) }

{ 定数宣言 }
const
    SCRIPTTBLMAX = 128;
    MOVESCRIPTTBLMAX = 128;
    PLAYERMOVESCRIPTTBLMAX = 8 + 1;
    SCRIPTTERMINATOR = -1;


{ 型宣言 }
type
    TScriptData = record        { TScriptData 構造体の定義      }
        x: integer;             { 出現する画面上の横位置        }
        y: integer;             { 出現する画面上の縦位置        }
        MonsterType: integer;   { 敵の種類                      }
        isnext: boolean;        { 次のデータをすぐに読み込むか? }
    end;
    TMoveScriptData = record    { TMoveScriptData 構造体の定義  }
        x: integer;             { 横の移動量                    }
        y: integer;             { 縦の移動量                    }
```

▼

List 2E-3

```
        MoveType: integer;      { 移動方法の種類              }
        Count: integer;         { 次のデータ読むまでのカウンタ      }
    end;
    TMoveScriptTbl = array[0..MOVESCRIPTTBLMAX] of TMoveScriptData;
    pTMoveScriptTbl = ^TMoveScriptTbl;


{ 定数宣言 }
const
    { 敵表示／イベント用スクリプトデータ    }
    { ゲーム終了か場面変更までデータがある }
    ScriptTbl: array[0..SCRIPTTBLMAX] of TScriptData = (
        (x: 60; y: 0; MonsterType: 0; isnext: false),
                          .
                         省略
                          .
        (x: 0; y: 0; MonsterType: SCRIPTTERMINATOR; isnext: false)
    );

    { 敵移動用スクリプトデータ      }
    { 敵の種類別に移動の最後までデータがある }
    Monster1MoveScriptTbl: array[0..MOVESCRIPTTBLMAX]
                                        of TMoveScriptData = (
        (x: 1; y: 1; MoveType: 0; Count: 0),
                          .
                         省略
                          .
        (x: 0; y: 0; MoveType: SCRIPTTERMINATOR; Count: 0)
    );

    { 自機移動用スクリプトデータ      }
    { 8方向 + 1 までデータがある    }
    { データは内部キーコードと対応 }
    PlayerMoveScriptTbl: array[0..PLAYERMOVESCRIPTTBLMAX]
                                        of TMoveScriptData = (
        (x:  0; y:  0; MoveType: 0; Count: 0),
        (x: -1; y: -1; MoveType: 0; Count: 0),
                          .
                         省略
                          .
        (x: 0; y: 0; MoveType: SCRIPTTERMINATOR; Count: 0)
    );
```

List 2E-4●スクリプトのデータ構造（C/C++）

```
/＊ 定数宣言 ＊/
#define SCRIPTTBLMAX              128
#define MOVESCRIPTTBLMAX          128
#define PLAYERMOVESCRIPTTBLMAX    8+1
#define SCRIPTTERMINATOR          -1

/＊ 型宣言 ＊/
typedef struct {          /＊ TScriptData 構造体の定義         ＊/
    int x;                /＊ 出現する画面上の横位置           ＊/
    int y;                /＊ 出現する画面上の縦位置           ＊/
    int MonsterType;      /＊ 敵の種類                         ＊/
    int isnext;           /＊ 次のデータをすぐに読み込むか?    ＊/
} TScriptData;

typedef struct {          /＊ TMoveScriptData 構造体の定義     ＊/
    int x;                /＊ 横の移動量                       ＊/
    int y;                /＊ 縦の移動量                       ＊/
    int MoveType;         /＊ 移動方法の種類                   ＊/
    int Count;            /＊ 次のデータ読むまでのカウンタ     ＊/
} TMoveScriptData;

typedef TMoveScriptData TMoveScriptTbl[MOVESCRIPTTBLMAX + 1];
typedef TMoveScriptTbl ＊pTMoveScriptTbl;

/＊ 定数宣言 ＊/
/＊ 敵表示／イベント用スクリプトデータ     ＊/
/＊ ゲーム終了か場面変更までデータがある ＊/
TScriptData ScriptTbl[SCRIPTTBLMAX + 1] = {
    60, 0, 0, false,
          ・
         省略
          ・
    0, 0, SCRIPTTERMINATOR, false,
};

/＊ 敵移動用スクリプトデータ      ＊/
/＊ 敵の種類別に移動の最後までデータがある ＊/
TMoveScriptData Monster1MoveScriptTbl[MOVESCRIPTTBLMAX + 1] = {
    1, 1, 0, 0,
        ・
       省略
        ・
    0, 0, SCRIPTTERMINATOR, 0,
};

/＊ 自機移動用スクリプトデータ      ＊/
/＊ 8方向までデータがある ＊/
TMoveScriptData PlayerMoveScriptTbl[PLAYERMOVESCRIPTTBLMAX + 1] = {
```

List 2E-4

```
     0,  0, 0, 0,
    -1, -1, 0, 0,
        .
      省略
        .
    0, 0, SCRIPTTERMINATOR, 0,
};
```

List 2E-5 ●キャラクタのデータ構造(Delphi)

```
{ 定数宣言 }
const
    { キャラクタの種類定義 }
    CHARACTERDATA       = 0;
    CHARACTERPLAYER     = 0;
    CHARACTERMONSTER1 = 1;
    CHARACTERMONSTER2 = 2;
    CHARACTERMONSTER3 = 3;
    CHARACTERDESTROY    = 4;
    MONSTERBULLETT      = 5;
    PLAYERBULLETT       = 6;
    CHARACTERTBLMAX     = 7;

    CHARACTERMOVETBLMAX = 100;
    MOVETBLMAX = 100;
    BULLETTBLMAX = 100;


{ 型宣言 }
type
    TCharacterData = record             { TCharacterData 構造体の定義  }
                                        { アニメーションパターン        }
        Pattern: array[0..PATTERNTBLMAX] of ^byte;
        Width: integer;                 { パターンの横サイズ          }
        Height: integer;                { パターンの縦サイズ          }
        MoveScriptTbl: pTMoveScriptTbl; { 移動方法用テーブルへのポインタ }
        AttackType: integer;            { 攻撃方法                }
        Score: LongInt;                 { 得点                  }
    end;
    TMoveData = record                  { TMoveData 構造体の定義     }
        SpOfs: integer;                 { スプライトテーブルの位置      }
        AttackX: integer;               { 当たり判定バッファの横位置     }
        AttackY: integer;               { 当たり判定バッファの縦位置     }
        AttackWidth: integer;           { 当たり判定バッファの横サイズ    }
        AttackHeight: integer;          { 当たり判定バッファの縦サイズ    }
        CharacterType: integer;         { キャラクタの種類           }
        MoveScriptCount: integer;       { 移動方法テーブル参照用        }
        AnimeCount: integer;            { アニメーションパターン用       }
```

```
      FlagEmpty: boolean;             { 空きフラグ                    }
    end;
    TMoveDataTbl = array[0..MOVETBLMAX] of TMoveData;
    pTMoveDataTbl = ^TMoveDataTbl;

{ 変数宣言 }
var
    { キャラクタデータのテーブル }
    { キャラクタの数だけデータがある 弾も含む }
    CharacterTbl: array[0..CHARACTERTBLMAX] of TCharacterData;

    { キャラクタ移動管理用のテーブル }
    CharacterMoveTbl: array[0..CHARACTERMOVETBLMAX] of TMoveData;

    { 自機／敵キャラクタ　弾移動管理用のテーブル }
    BulletMoveTbl: array[0..BULLETTBLMAX] of TMoveData;
```

List 2E-6 ●キャラクタのデータ構造(C/C++)

```
/* 定数宣言 */
/* キャラクタの種類定義 */
#define CHARACTERPLAYER      0
#define CHARACTERMONSTER1    1
#define CHARACTERMONSTER2    2
#define CHARACTERMONSTER3    3
#define CHARACTERDESTROY     4
#define MONSTERBULLETT       5
#define PLAYERBULLETT        6
#define CHARACTERTBLMAX      7

#define CHARACTERMOVETBLMAX  100
#define MOVETBLMAX           100
#define BULLETTBLMAX         100

/* 型宣言 */
typedef struct {                      /* TCharacterData 構造体の定義    */
                                      /* アニメーションパターン          */
    unsigned char *Pattern[PATTERNTBLMAX + 1];
    int Width;                        /* パターンの横サイズ              */
    int Height;                       /* パターンの縦サイズ              */
    TMoveScriptTbl *MoveScriptTbl;    /* 移動方法用テーブルへのポインタ  */
    int AttackType;                   /* 攻撃方法                        */
    long Score;                       /* 得点                            */
} TCharacterData;

typedef struct {                      /* TMonsterMoveData 構造体の定義 */
    int SpOfs;                        /* スプライトテーブルの位置        */
    int AttackX;                      /* 当たり判定バッファの横位置      */
```

List 2E-6

```
    int AttackY;                       /* 当たり判定バッファの縦位置        */
    int AttackWidth;                   /* 当たり判定バッファの横サイズ      */
    int AttackHeight;                  /* 当たり判定バッファの縦サイズ      */
    int CharacterType;                 /* キャラクタの種類                  */
    int MoveScriptCount;               /* 移動方法テーブル参照用            */
    int AnimeCount;                    /* アニメーションパターン用          */
    int FlagEmpty;                     /* 空きフラグ                        */
  } TMoveData;
  typedef TMoveData TMoveDataTbl[MOVETBLMAX + 1];
  typedef TMoveDataTbl *pTMoveDataTbl;

  /* 変数宣言 */

  /* キャラクタデータのテーブル */
  /* キャラクタの数だけデータがある 弾も含む */
  TCharacterData CharacterTbl[CHARACTERTBLMAX + 1];

  /* キャラクタ移動管理用のテーブル */
  TMoveData CharacterMoveTbl[CHARACTERMOVETBLMAX + 1];

  /* 自機／敵キャラクタ　敵キャラクタ弾移動管理用のテーブル */
  TMoveData BulletMoveTbl[BULLETTBLMAX + 1];
```

List 2E-7●シューティングゲームのアルゴリズム(Delphi)

```
  { 縦スクロールタイプ                             }
  { 背景書換は２つのバッファを使う方法を使っています }
  { 当たり判定は移動量ごとに区切られた２次元配列から }
  { 得る方法です。                                       }

  (* ------------------------------------------------ *)
  { 定数宣言 }
  const
    MapViewWidthDefLimit  = 16;     { 画面の横幅                }
    MapViewHeightDefLimit = 14;     { 画面の縦幅                }
    MapPartssize          = 32;     { パーツの dot 数（縦横共通）}
    MapWidthLimit         = 16;     { マップの横幅              }
    MapHeightLimit        = 1000;   { マップの縦幅              }

    ScrollCopyLineCountLimit = 4;
    MoveCount = 8;

    { 当たり判定用バッファのサイズ }
    MapAttackbufWidthLimit  = (MapViewWidthDefLimit * MapPartssize)
                                                    div MoveCount;
    MapAttackbufHeightLimit = (MapViewHeightDefLimit * MapPartssize)
                                                    div MoveCount;

  (* ------------------------------------------------ *)
  { 背景処理 }
```

```
var
  ScrollMapY, ScrollViewX, ScrollCopyLineCount: integer;

{ 全画面表示 }
{ 縦方向に限定しているので MapX は常に 0 }
procedure MapView(MapX, MapY, ViewX, ViewY: integer);
var
   tmpofs : point_t;
   cellp : cell_ptr;
    x, y, TmpX, TmpY: integer;
begin
    TmpX := MapX;
    TmpY := MapY;
    ScrollMapY := MapY;
    ScrollViewX := ViewX;
    for y := 0 to ViewY-1 do begin
        for x := 0 to ViewX-1 do begin
            cellp := rpg_map_getcell(@tmpofs);{ セルを取得            }
            vbuf_copy(@tmpofs, cellp, MapPartssize);
                                            { 画面バッファへコピー  }
            inc(TmpX);
        end;
        TmpX := MapX;    { x を基点に戻す }
        inc(TmpY);
    end;
end;

{ 背景 1パーツラインぶん取得                                  }
{ 縦方向に限定しているので MapX は常に 0                      }
{ 縦方向に1パーツライン取る場合 LineX = 画面パーツサイズ, LineY = 1 }
procedure MapGetLine(MapX, MapY, LineX, LineY: integer);
var
   tmpofs : point_t;
   cellp : cell_ptr;
    x, y, TmpX, TmpY: integer;
begin
    TmpX := MapX;
    TmpY := MapY;
    for y := 0 to LineY-1 do begin
        for x := 0 to LineX-1 do begin
            cellp := MapGetcell(@tmpofs);            { セルを取得    }
            { ラインバッファへコピー }
            linebufcopy(@tmpofs, cellp, MapPartssize);
            inc(TmpX);
        end;
        TmpX := MapX;                              { x を基点に戻す }
        inc(TmpY);
    end;
end;
```

List 2E-7

```
{ 背景スクロール }
{ 移動方向は上に固定 }
procedure MapScrollUp;
begin
    { コピーしたライン数をチェック }
    if ScrollCopyLineCount = ScrollCopyLineCountLimit then begin
        if ScrollMapY = 0 then
            exit;
        { 新しく1パーツライン取得 }
        Dec(ScrollMapY);
        MapGetLine(0, ScrollMapY, ScrollViewX, 1);
        ScrollCopyLineCount := 0;
    end else begin
        inc(ScrollCopyLineCount);
    end;
    { 画面バッファをスクロールラインずらす }
    { 両方のバッファでスクロール }
    vbuf1Scroll(ScrollCopyLine);
    vbuf2Scroll(ScrollCopyLine);
    { 画面バッファへスクロールラインをコピー  }
    vbuf1LineCopy(ScrollCopyLineCount, ScrollCopyLine);
    vbuf2LineCopy(ScrollCopyLineCount, ScrollCopyLine);
end;

(* ----------------------------------------------- *)
{ スプライト処理 }

{ レジスタの空きを探す }
function SpGetEmptyReg: integer;
var
    i: integer;
begin
    for i := 0 to SPREGTBLMAX-1 do begin
        if SpRegTbl[i].FlagEmpty then begin
            result := i;
            exit;
        end;
    end;
    result := -1;
end;

{ レジスタにデータを登録 }
function SpSetReg(SrcSpReg: TSpReg): integer;
var
    SpRegOfs: integer;
begin
    SpRegOfs := SpGetEmptyReg;
    result := SpRegOfs;
    if SpRegOfs = -1 then
```

```
        exit;
    Move(SrcSpReg, SpRegTbl[SpRegOfs], Sizeof(TSpReg));
    SpRegTbl[SpRegOfs].FlagEmpty := False;
end;

{ レジスタのデータを削除 }
procedure SpFreeReg(SpOfs: integer);
begin
    SpRegTbl[SpOfs].FlagEmpty := True;
end;

{ 指定したレジスタのグラフィックデータを消去 }
procedure SpClearDraw(SpOfs: integer);
begin
    if not SpRegTbl[SpOfs].FlagEmpty then begin
        { 背景画面バッファから画面バッファへデータサイズだけコピー }
        { 優先順位がいちばん上以外なら消去しません                }
        if SpRegTbl[SpOfs].priority <> 0 then
            exit;
        vbuf1to2Draw(SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
          SpRegTbl[SpOfs].Width, SpRegTbl[SpOfs].Height);
    end;
end;

{ 指定したレジスタをスプライト画面に合成 }
procedure SpRopDraw(SpOfs: integer);
begin
    if not SpRegTbl[SpOfs].FlagEmpty then begin
        { スプライト画面に合成 }
        { 合成時に効果を加えるときはこの段階で行う }
        vbuf2RopDraw(SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
          SpRegTbl[SpOfs].Width, SpRegTbl[SpOfs].Height,
          SpRegTbl[SpOfs].gra);
    end;
end;

{ 全レジスタを検索してスプライト画面に合成 }
{ 優先順位は無視しています }
procedure SpRopDrawAll;
var
    i: integer;
begin
    for i := 0 to SPREGTBLMAX-1 do begin
        SpRopDraw(i);
    end;
end;

{ 全レジスタを検索してスプライト画面に合成 }
{ 優先順位4段階までありで、              }
{ いったんソートをさせてます             }
```

List 2E-7

```
{ より速くするならスプライトレジスタ登録    }
{ 時にソートテーブルに入れるようにします    }
procedure SpPriorityRopDrawAll;
var
    SpSortTbl: array[0..3, 0..SPREGTBLMAX] of integer;
    SpSortTblLimit: array[0..3] of integer;
    i, priority, SortLimit: integer;
begin
    for priority := 0 to 3 do
        SpSortTblLimit[priority] := 0;
    for i := 0 to SPREGTBLMAX-1 do begin
        if not SpRegTbl[i].FlagEmpty then begin
            priority := SpRegTbl[i].priority;
            SortLimit := SpSortTblLimit[priority];
            SpSortTbl[priority][SortLimit] := i;
            Inc(SpSortTblLimit[priority]);
        end;
    end;
    for priority := 0 to 3 do begin
        for i := 0 to SpSortTblLimit[priority]-1 do begin
            SpRopDraw(SpSortTbl[priority][i]);
        end;
    end;
end;

(* ------------------------------------------------ *)
{ 当たり判定 }

type
  TAttackbufData = record       { TAttackbufData 構造体の定義  }
    CharacterType: integer;     { キャラクタ種別              }
    TblOfs: integer;            { データテーブルの位置         }
  end;

var
  { 当たり判定検出用バッファ }
  Attackbuf: array[0..MapAttackbufWidthLimit,
                  0..MapAttackbufHeightLimit] of TAttackbufData;

const
  AttackbufEmpty   = 0;
  AttackbufMonster = 1;
  AttackbufPlayer  = 2;

{ 画面座標から当たり判定バッファの座標に変換 }
procedure SetAttackVal(x, y, Width, Height: integer;
  var SrcMoveData: TMoveData);
begin
    with SrcMoveData do begin
        AttackX := x div MoveCount;
```

```
        AttackY := y div MoveCount;
        AttackWidth := Width div MoveCount;
        AttackHeight := Height div MoveCount;
    end;
end;

{ 当たり判定バッファに登録 }
procedure AddAttackBuf(CharacterType,
                              TblOfs, x, y, Width, Height: integer;
  var SrcMoveData: TMoveData);
var
  TmpX, TmpY, TmpWidth, TmpHegiht: integer;
begin
    SetAttackVal(x, y, Width, Height, SrcMoveData);
    with SrcMoveData do begin
        TmpX := AttackX;
        TmpY := AttackY;
    end;
    for TmpHegiht := 0 to SrcMoveData.AttackHeight do begin
        for TmpWidth := 0 to SrcMoveData.AttackWidth do begin
            Attackbuf[TmpX][TmpY].CharacterType := CharacterType;
            Attackbuf[TmpX][TmpY].TblOfs := TblOfs;
            inc(TmpX);
        end;
        TmpX := SrcMoveData.AttackX;
        inc(TmpY);
    end;
end;

{ 当たり判定バッファから削除 }
procedure FreeAttackBuf(var SrcMoveData: TMoveData);
var
  TmpX, TmpY, TmpWidth, TmpHegiht: integer;
begin
    with SrcMoveData do begin
        TmpX := AttackX;
        TmpY := AttackY;
    end;
    for TmpHegiht := 0 to SrcMoveData.AttackHeight do begin
        for TmpWidth := 0 to SrcMoveData.AttackWidth do begin
            Attackbuf[TmpX][TmpY].CharacterType := AttackbufEmpty;
            Attackbuf[TmpX][TmpY].TblOfs := 0;
        end;
        TmpX := SrcMoveData.AttackX;
        inc(TmpY);
    end;
end;

{ 弾の当たり判定 }
function isBulletAttack(x, y, AttackbufType: integer): boolean;
```

List 2E-7

```
begin
    if (Attackbuf[x][y].CharacterType = AttackbufType) then begin
        result := True;
    end else begin
        result := False;
    end;
end;

(* --------------------------------------------- *)
{ キャラクタ操作 }

const
    { 最初に射出された弾をどれだけ自機と離すか }
    PlayerBulletAddX = 8;
    PlayerBulletAddY = 0;

{ 移動用テーブルの空きを探す }
function GetTblEmpty
        (TblLimit: integer; SrcMoveTbl: pTMoveDataTbl): integer;
var
    i: integer;
begin
    for i := 0 to TblLimit-1 do begin
        if SrcMoveTbl^[i].FlagEmpty then begin
            result := i;
            exit;
        end;
    end;
    result := -1;
end;

{ キャラクタ登録 }
procedure CharacterCreate(CharacterType, x, y,
                    TblLimit: integer; SrcMoveTbl: pTMoveDataTbl);
var
    SrcSpReg: TSpReg;
    TblOfs, SpOfs, AttackBufVal: integer;
begin
    TblOfs := GetTblEmpty(TblLimit, SrcMoveTbl);
    if TblOfs = -1 then
        exit;
    { スプライトレジスタ登録 }
    SrcSpReg.x := x;
    SrcSpReg.y := y;
    SrcSpReg.gra := GetInitAnimePattern(CharacterType);
    SrcSpReg.mode := 0;
    SrcSpReg.priority := 0;
    SpOfs := SpSetReg(SrcSpReg);
    if SpOfs = -1 then
        exit;
```

```
    { 移動用テーブル登録 }
    SrcMoveTbl^[TblOfs].SpOfs := SpOfs;
    SrcMoveTbl^[TblOfs].CharacterType := CharacterType;
    SrcMoveTbl^[TblOfs].AnimeCount := 0;
    SrcMoveTbl^[TblOfs].MoveScriptCount := 0;
    SrcMoveTbl^[TblOfs].FlagEmpty := False;
    { 弾を登録するときは当たり判定に使う座標だけを設定 }
    if (CharacterType = MONSTERBULLETT) or
                    (CharacterType = PLAYERBULLETT) then begin
        { 当たり判定用座標を設定 }
        SetAttackVal(x, y,
          CharacterTbl[CharacterType].Width,
          CharacterTbl[CharacterType].Height,
          BulletMoveTbl[TblOfs]);
    end else begin
        { 当たり判定用テーブル登録 }
        if CharacterType = CHARACTERPLAYER then begin
            AttackBufVal := AttackbufPlayer;
        end else begin
            AttackBufVal := AttackbufMonster;
        end;
        AddAttackBuf(AttackBufVal, TblOfs, x, y,
          CharacterTbl[CharacterType].Width,
          CharacterTbl[CharacterType].Height,
          CharacterMoveTbl[TblOfs]);
    end;
end;

{ 弾の当たり判定をする }
procedure CharacterAttackCheck(CharacterType: integer;
  var SrcMoveData: TMoveData);
var
    AttackBufVal: integer;
begin
    if (CharacterType = PLAYERBULLETT) or
                  (CharacterType = CHARACTERPLAYER) then begin
        AttackBufVal := AttackbufMonster;
    end else begin
        AttackBufVal := AttackbufPlayer;
    end;
    if isBulletAttack(SrcMoveData.AttackX,
                  SrcMoveData.AttackY, AttackBufVal) then begin
        if CharacterType = PLAYERBULLETT then begin
            { 自機が撃った弾が敵に当った }
            { 爆発したシーンに切り換え }
            CharacterMonsterDestroy(SrcMoveData.AttackX,
                                        SrcMoveData.AttackY);
            { 弾データの消去 }
            SpFreeReg(SrcMoveData.SpOfs);
            SrcMoveData.FlagEmpty := True;
```

List 2E-7

```
        end else if CharacterType = CHARACTERPLAYER then begin
            { 自機が敵キャラクタに当った }
            { 自機が爆発したシーンなどに切り換え }
            CharacterPlayerDestroy(SrcMoveData.AttackX,
                                   SrcMoveData.AttackY);
            { スプライトデータなどを爆発したシーンに切り換え }
            CharacterMonsterDestroy(SrcMoveData.AttackX,
                                   SrcMoveData.AttackY);
        end else begin
            { 敵キャラクタが撃った弾が自機に当った }
            { 自機が爆発したシーンなどに切り換え }
            CharacterPlayerDestroy(SrcMoveData.AttackX,
                                   SrcMoveData.AttackY);
            { 弾データの消去 }
            SpFreeReg(SrcMoveData.SpOfs);
            SrcMoveData.FlagEmpty := True;
        end;
    end;
end;

{ キャラクタ移動 }
procedure CharacterMove(TblOfs: integer;
  var SrcMoveData: TMoveData);
var
    SpOfs, CharacterType: integer;
begin
    if not SrcMoveData.FlagEmpty then begin
        SpOfs := SrcMoveData.SpOfs;
        { 画面上のキャラクタを消す }
        SpClearDraw(SpOfs);
        { 移動方法スクリプトが終わったかどうか }
        if isMoveScriptEnd(SrcMoveData) then begin
            { 終わったなら確保したデータを消す }
            SpFreeReg(SpOfs);
            SrcMoveData.FlagEmpty := True;
            continue;
        end;
        { 移動距離をスプライトレジスタに反映 }
        { 移動距離を示したカウンタも更新     }
        SetMoveScriptToSp(SrcMoveData, SpOfs);
        { アニメーションパターン取得 }
        CharacterType := SrcMoveData.CharacterType;
        SpRegTbl[SpOfs].gra := GetAnimePattern(CharacterType,
                               SrcMoveData.AnimeCount);
        { 弾／自機のときは当たり判定を行う }
        if (CharacterType = MONSTERBULLETT) or
          (CharacterType = CHARACTERPLAYER) or
          (CharacterType = PLAYERBULLETT) then begin
            { 当たり判定用座標を設定 }
            SetAttackVal(SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
```

```
          CharacterTbl[CharacterType].Width,
          CharacterTbl[CharacterType].Height,
          SrcMoveData);
        { 当たり判定 }
        CharacterAttackCheck(CharacterType, SrcMoveData);
        if CharacterType = CHARACTERPLAYER then begin
          { 自機のときはさらに当たり判定バッファを更新 }
          FreeAttackBuf(SrcMoveData);
          AddAttackBuf(AttackbufPlayer, TblOfs,
            SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
            CharacterTbl[CharacterType].Width,
            CharacterTbl[CharacterType].Height,
            SrcMoveData);
        end;
      end else begin
        { 当たり判定バッファを更新 }
        FreeAttackBuf(SrcMoveData);
        AddAttackBuf(AttackbufMonster, TblOfs,
          SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
          CharacterTbl[CharacterType].Width,
          CharacterTbl[CharacterType].Height,
          SrcMoveData);
      end;
    end;
end;

{ 指定されたテーブルのキャラクタを全部移動 }
procedure CharacterMoveAll
        (TblLimit: integer; SrcMoveTbl: pTMoveDataTbl);
var
  i: integer;
begin
    for i := 0 to TblLimit-1 do begin
      CharacterMove(i, SrcMoveTbl^[i]);
    end;
end;

{ 敵の弾登録 }
procedure MonsterBulletCreate(x, y: integer);
begin
    CharacterCreate(MONSTERBULLETT, x, y, BULLETTBLMAX,
                                              @BulletMoveTbl);
end;

{ 自機の弾登録 }
procedure PlayerBulletCreate;
var
    SpOfs: integer;
begin
    SpOfs := CharacterMoveTbl[CHARACTERPLAYER].SpOfs;
```

List 2E-7

```
    CharacterCreate(PLAYERBULLETT,
      SpRegTbl[SpOfs].x + PlayerBulletAddX,
      SpRegTbl[SpOfs].y + PlayerBulletAddY,
      BULLETTBLMAX, @BulletMoveTbl);
end;

{ 敵の弾移動 }
procedure MonsterBulletMove;
begin
    CharacterMoveAll(BULLETTBLMAX, @BulletMoveTbl);
end;

{ 自機の弾移動 }
procedure PlayerBulletMove;
begin
    CharacterMoveAll(BULLETTBLMAX, @BulletMoveTbl);
end;

(* ----------------------------------------------- *)
{ タイマから呼び出される変数 }
procedure TimerEntry;
begin
    { 背景スクロール }
    MapScrollUp;
    { 画面上のキャラクタ移動 }
    { 移動とともに画面も書き換える }
    CharacterMoveAll(MOVETBLMAX, @CharacterMoveTbl);
    { 弾移動 }
    PlayerBulletMove;
    MonsterBulletMove;
end;

(* ----------------------------------------------- *)
{ 自機の移動と弾の発射 }

const
  KEYDIRECTION_NOTMOVE    = 0;
  KEYDIRECTION_UPLEFT     = 1;
  KEYDIRECTION_UPCENTER   = 2;
  KEYDIRECTION_UPRIGHT    = 3;
  KEYDIRECTION_RIGHT      = 4;
  KEYDIRECTION_LEFT       = 5;
  KEYDIRECTION_DOWNLEFT   = 6;
  KEYDIRECTION_DOWNCENTER = 7;
  KEYDIRECTION_DOWNRIGHT  = 8;
  KEYTRIGGERA = 9;
  KEYTRIGGERB = 10;

var
   PlayerMoveTblOfs: integer;
```

```
{ 自機移動 }
{ キーは同時入力できるように、この関数を呼びだすところで    }
{ 内部のキーコード(上記の定数)にまとめておきます          }
procedure PlayerMove(Key: integer);
begin
    if (Key = KEYTRIGGERA) or (Key = KEYTRIGGERB) then begin
        { 弾の射出 }
        PlayerBulletCreate;
    end else begin
        { 自機移動 }
        CharacterMoveTbl[PlayerMoveTblOfs].MoveScriptCount := Key;
        CharacterMove(PlayerMoveTblOfs,
                         CharacterMoveTbl[PlayerMoveTblOfs]);
        CharacterMoveTbl[PlayerMoveTblOfs].
                           MoveScriptCount := KEYDIRECTION_NOTMOVE;
    end;
end;
```

List 2E-8●シューティングゲームのアルゴリズム(C/C++)

```
/* 縦スクロールタイプ                                      */
/* 背景書換は2つのバッファを使う方法を使っています          */
/* 当たり判定は移動量ごとに区切られた2次元配列から          */
/* 得る方法です。                                          */

/* -------------------------------------------- */
/* 定数宣言 */
#define MapViewWidthDefLimit  16    /* 画面の横幅              */
#define MapViewHeightDefLimit 14    /* 画面の縦幅              */
#define MapPartssize          32
                                    /* パーツの dot 数 (縦横共通) */
#define MapWidthLimit         16    /* マップの横幅            */
#define MapHeightLimit        1000  /* マップの縦幅            */

#define ScrollCopyLineCountLimit  4
#define MoveCount   8

  /* 当たり判定用バッファのサイズ */
#define MapAttackbufWidthLimit
        ((MapViewWidthDefLimit * MapPartssize) / MoveCount)
#define MapAttackbufHeightLimit
        ((MapViewHeightDefLimit * MapPartssize) / MoveCount)

/* ----------------------------------------------- */
/* 背景処理 */

static int ScrollMapY, ScrollViewX, ScrollCopyLineCount;
```

List 2E-8

```
/* 全画面表示 */
/* 縦方向に限定しているので MapX は常に 0 */
void MapView(int MapX, int MapY, int ViewX, int ViewY)
{
    point_t tmpofs;
    cell_t *cellp;
    int x, y, TmpX, TmpY;

    TmpX = MapX;
    TmpY = MapY;
    ScrollMapY = MapY;
    ScrollViewX = ViewX;
    for (y = 0; y < ViewY; y++, TmpY++) {
        for (x = 0; x < ViewX; x++, TmpX++) {
            cellp = MapGetcell(&tmpofs);    /* セルを取得      */
            vbuf_copy(&tmpofs, cellp, &partssize);
                                           /* 画面バッファへコピー */
        }
        TmpX = MapX;                       /* x を基点に戻す    */
    }
}

/* 背景 1パーツライン分取得 */
/* 縦方向に限定しているので MapX は常に 0                     */
/* 縦方向に1パーツライン取る場合                              */
/*  LineX = 画面パーツサイズ, LineY = 1                       */
void MapGetLine(int MapX, int MapY, int LineX, int LineY)
{
    point_t tmpofs;
    cell_t *cellp;
    int x, y, TmpX, TmpY;

    TmpX = MapX;
    TmpY = MapY;
    for (y = 0; y < LineY; y++, TmpY++) {
        for (x = 0; x < LineX; x++, TmpX++) {
            cellp = MapGetcell(&tmpofs);    /* セルを取得    */
            /* ラインバッファへコピー */
            linebufcopy(&tmpofs, cellp, MapPartssize);
        }
        TmpX = MapX;                          /* x を基点に戻す  */
    }
}

/* 背景スクロール */
/* 移動方向は上に固定 */
void MapScrollUp(void)
{
    /* コピーしたライン数をチェック */
    if (ScrollCopyLineCount == ScrollCopyLineCountLimit) {
```

```
        if (ScrollMapY == 0)
            return;
        /* 新しく1パーツライン取得 */
        ScrollMapY--;
        MapGetLine(0, ScrollMapY, ScrollViewX, 1);
        ScrollCopyLineCount = 0;
    } else {
        ScrollCopyLineCount++;
    }
    /* 画面バッファをスクロールラインずらす */
    /* 両方のバッファでスクロール */
    vbuf1Scroll(ScrollCopyLine);
    vbuf2Scroll(ScrollCopyLine);
    /* 画面バッファヘスクロールラインをコピー  */
    vbuf1LineCopy(ScrollCopyLineCount, ScrollCopyLine);
    vbuf2LineCopy(ScrollCopyLineCount, ScrollCopyLine);
}

/* ----------------------------------------------- */
/* スプライト処理 */

/* レジスタの空きを探す */
int SpGetEmptyReg(void)
{
    int i;

    for (i = 0; i < SPREGTBLMAX; i++) {
        if (SpRegTbl[i].FlagEmpty) {
            return i;
        }
    }
    return -1;
}

/* レジスタにデータを登録 */
int SpSetReg(TSpReg SrcSpReg)
{
    int SpRegOfs;

    SpRegOfs = SpGetEmptyReg();
    if (SpRegOfs = -1)
        return -1;
    memcpy(&SpRegTbl[SpRegOfs], &SrcSpReg, sizeof(SrcSpReg));
    SpRegTbl[SpRegOfs].FlagEmpty = False;
    return SpRegOfs;
}

/* レジスタのデータを削除 */
void SpFreeReg(int SpOfs)
{
    SpRegTbl[SpOfs].FlagEmpty = True;
```

List 2E-8

```
}

/* 指定したレジスタのグラフィックデータを消去 */
void SpClearDraw(int SpOfs)
{
    if (!(SpRegTbl[SpOfs].FlagEmpty)) {
        /* 背景画面バッファから画面バッファへデータサイズ  */
        /* だけコピー                                      */
        /* 優先順位がいちばん上以外なら消去しません        */
        if (SpRegTbl[SpOfs].priority != 0)
            return;
        vbuf1to2Draw(SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
          SpRegTbl[SpOfs].Width, SpRegTbl[SpOfs].Height);
    }
}

/* 指定したレジスタをスプライト画面に合成 */
void SpRopDraw(int SpOfs)
{
    if (!(SpRegTbl[SpOfs].FlagEmpty)) {
        /* スプライト画面に合成 */
        /* 合成時に効果を加えるときはこの段階で行う */
        vbuf2RopDraw(SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
          SpRegTbl[SpOfs].Width, SpRegTbl[SpOfs].Height,
          SpRegTbl[SpOfs].gra);
    }
}

/* 全レジスタを検索してスプライト画面に合成 */
/* 優先順位は無視しています */
void SpRopDrawAll(void)
{
    int i;

    for (i = 0; i < SPREGTBLMAX; i++) {
        SpRopDraw(i);
    }
}

/* 全レジスタを検索してスプライト画面に合成 */
/* 優先順位4段階までありで、               */
/* いったんソートをさせてます              */
/* より速くするならスプライトレジスタ登録  */
/* 時にソートテーブルに入れるようにします  */
void SpPriorityRopDrawAll(void)
{
    int SpSortTbl[4 + 1][SPREGTBLMAX + 1];
    int SpSortTblLimit[4 + 1];
    int i, priority, SortLimit;

    for (priority = 0; priority < 4; priority++)
```

```
        SpSortTblLimit[priority] = 0;
    for (i = 0; i < SPREGTBLMAX; i++) {
        if (!(SpRegTbl[i].FlagEmpty)) {
            priority = SpRegTbl[i].priority;
            SortLimit = SpSortTblLimit[priority];
            SpSortTbl[priority][SortLimit] = i;
            SpSortTblLimit[priority]++;
        }
    }
    for (priority = 0; priority < 4; priority++) {
        for (i = 0; i < SpSortTblLimit[priority]-1; i++) {
            SpRopDraw(SpSortTbl[priority][i]);
        }
    }
}

/* ----------------------------------------------- */
/* 当たり判定 */

typedef struct {            /* TAttackbufData 構造体の定義   */
    int CharacterType;      /* キャラクタ種別                */
    int TblOfs;             /* データテーブルの位置          */
} TAttackbufData;

/* 当たり判定検出用バッファ */
TAttackbufData Attackbuf[MapAttackbufWidthLimit + 1]
                        [MapAttackbufHeightLimit + 1];

#define AttackbufEmpty   0
#define AttackbufMonster 1
#define AttackbufPlayer  2


/* 画面座標から当たり判定バッファの座標に変換 */
void SetAttackVal(int x, int y, int Width, int Height,
                                 TMoveData *SrcMoveData)
{
    SrcMoveData->AttackX = x / MoveCount;
    SrcMoveData->AttackY = y / MoveCount;
    SrcMoveData->AttackWidth = Width / MoveCount;
    SrcMoveData->AttackHeight = Height / MoveCount;
}

/* 当たり判定バッファに登録 */
void AddAttackBuf(int CharacterType, int TblOfs, int x, int y,
                  int Width, int Height, TMoveData *SrcMoveData)
{
    int TmpX, TmpY, TmpWidth, TmpHegiht;

    SetAttackVal(x, y, Width, Height, SrcMoveData);
    TmpX = SrcMoveData->AttackX;
```

List 2E-8

```
    TmpY = SrcMoveData->AttackY;
    for (TmpHegiht = 0, TmpY = 0; TmpHegiht <= SrcMoveData->
                                AttackHeight; TmpY++, TmpHegiht++) {
        for (TmpWidth = 0, TmpX = SrcMoveData->AttackX; TmpWidth <=
                         SrcMoveData->AttackWidth; TmpX++, TmpWidth++) {
            Attackbuf[TmpX][TmpY].CharacterType = CharacterType;
            Attackbuf[TmpX][TmpY].TblOfs = TblOfs;
        }
    }
}

/* 当たり判定バッファから削除 */
void FreeAttackBuf(TMoveData *SrcMoveData)
{
    int TmpX, TmpY, TmpWidth, TmpHegiht;

    TmpX = SrcMoveData->AttackX;
    TmpY = SrcMoveData->AttackY;
    for (TmpHegiht = 0, TmpY = 0; TmpHegiht <= SrcMoveData->
                                AttackHeight; TmpY++, TmpHegiht++) {
        for (TmpWidth = 0, TmpX = SrcMoveData->AttackX; TmpWidth <=
                       SrcMoveData->AttackWidth; TmpX++, TmpWidth++) {
            Attackbuf[TmpX][TmpY].CharacterType = AttackbufEmpty;
            Attackbuf[TmpX][TmpY].TblOfs = 0;
        }
    }
}

/* 弾の当たり判定 */
int isBulletAttack(int x, int y, int AttackbufType)
{
    if (Attackbuf[x][y].CharacterType == AttackbufType) {
        return True;
    }
    return False;
}

/* ----------------------------------------------- */
/* キャラクタ操作 */

/* 最初に射出された弾をどれだけ自機と離すか */
#define PlayerBulletAddX  8
#define PlayerBulletAddY  0

/* 移動用テーブルの空きを探す */
int GetTblEmpty(int TblLimit, TMoveDataTbl *SrcMoveTbl)
{
    int i;

    for (i = 0; i < TblLimit; i++) {
        if (SrcMoveTbl[i]->FlagEmpty) {
```

```
            return i;
        }
    }
    return -1;
}

/* キャラクタ登録 */
void CharacterCreate(int CharacterType, int x, int y,
                int TblLimit, TMoveDataTbl *SrcMoveTbl)
{
    TSpReg SrcSpReg;
    int TblOfs, SpOfs, AttackBufVal;

    TblOfs = GetTblEmpty(TblLimit, SrcMoveTbl);
    if (TblOfs == -1)
        return;
    /* スプライトレジスタ登録 */
    SrcSpReg.x = x;
    SrcSpReg.y = y;
    SrcSpReg.gra = GetInitAnimePattern(CharacterType);
    SrcSpReg.mode = 0;
    SrcSpReg.priority = 0;
    SpOfs = SpSetReg(SrcSpReg);
    if (SpOfs == -1)
        return;
    /* 移動用テーブル登録 */
    SrcMoveTbl[TblOfs]->SpOfs = SpOfs;
    SrcMoveTbl[TblOfs]->CharacterType = CharacterType;
    SrcMoveTbl[TblOfs]->AnimeCount = 0;
    SrcMoveTbl[TblOfs]->MoveScriptCount = 0;
    SrcMoveTbl[TblOfs]->FlagEmpty = False;
    /* 弾を登録するときは当たり判定に使う座標だけを設定 */
    if ((CharacterType == MONSTERBULLETT) ||
                        (CharacterType == PLAYERBULLETT)) {
        /* 当たり判定用座標を設定 */
        SetAttackVal(x, y,
          CharacterTbl[CharacterType].Width,
          CharacterTbl[CharacterType].Height,
          &BulletMoveTbl[TblOfs]);
    } else {
        /* 当たり判定用テーブル登録 */
        if (CharacterType == CHARACTERPLAYER) {
            AttackBufVal = AttackbufPlayer;
        } else {
            AttackBufVal = AttackbufMonster;
        }
        AddAttackBuf(AttackBufVal, TblOfs, x, y,
          CharacterTbl[CharacterType].Width,
          CharacterTbl[CharacterType].Height,
          &CharacterMoveTbl[TblOfs]);
    }
```

List 2E-8

```
}

/* 弾の当たり判定をする */
void CharacterAttackCheck(int CharacterType,
                              TMoveData *SrcMoveData)
{
    int AttackBufVal;

    if ((CharacterType == PLAYERBULLETT) ||
                (CharacterType == CHARACTERPLAYER)) {
        AttackBufVal = AttackbufMonster;
    } else {
        AttackBufVal = AttackbufPlayer;
    }
    if (isBulletAttack(SrcMoveData->AttackX,
                SrcMoveData->AttackY, AttackBufVal)) {
        if (CharacterType == PLAYERBULLETT) {
            /* 自機が撃った弾が敵に当った */
            /* 爆発したシーンに切り換え */
            CharacterMonsterDestroy(SrcMoveData->AttackX,
                                    SrcMoveData->AttackY);
            /* 弾データの消去 */
            SpFreeReg(SrcMoveData->SpOfs);
            SrcMoveData->FlagEmpty = True;
        } else if (CharacterType == CHARACTERPLAYER) {
            /* 自機が敵キャラクタに当った */
            /* 自機が爆発したシーンなどに切り換え */
            CharacterPlayerDestroy(SrcMoveData->AttackX,
                                   SrcMoveData->AttackY);
            /* 敵が爆発したシーンに切り換え */
            CharacterMonsterDestroy(SrcMoveData->AttackX,
                                    SrcMoveData->AttackY);
        } else {
            /* 敵キャラクタが撃った弾が自機に当った */
            /* 自機が爆発したシーンなどに切り換え */
            CharacterPlayerDestroy(SrcMoveData->AttackX,
                                   SrcMoveData->AttackY);
            /* 弾データの消去 */
            SpFreeReg(SrcMoveData->SpOfs);
            SrcMoveData->FlagEmpty = True;
        }
    }
}

/* キャラクタ移動 */
void CharacterMove(int TblOfs, TMoveData *SrcMoveData)
{
    int SpOfs, CharacterType;

    if (!(SrcMoveData->FlagEmpty)) {
        SpOfs = SrcMoveData->SpOfs;
```

```
        /* 画面上のキャラクタを消す */
        SpClearDraw(SpOfs);
        /* 移動方法スクリプトが終わったかどうか */
        if (isMoveScript(SrcMoveData)) {
            /* 終わったなら確保したデータを消す */
            SpFreeReg(SpOfs);
            SrcMoveData->FlagEmpty = True;
            continue;
        }
        /* 移動距離をスプライトレジスタに反映 */
        /* 移動距離を示したカウンタも更新      */
        SetMoveScriptToSp(SrcMoveData, SpOfs);
        /* アニメーションパターン取得 */
        CharacterType = SrcMoveData->CharacterType;
        SpRegTbl[SpOfs].gra = GetAnimePattern(CharacterType,
                                    SrcMoveData->AnimeCount);
        /* 弾のときは当たり判定を行う */
        if ((CharacterType == MONSTERBULLETT) ||
                    (CharacterType == CHARACTERPLAYER) ||
                    (CharacterType == PLAYERBULLETT)) {
            /* 当たり判定用座標を設定 */
            SetAttackVal(SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
              CharacterTbl[CharacterType].Width,
              CharacterTbl[CharacterType].Height,
              SrcMoveData);
            /* 当たり判定 */
            CharacterAttackCheck(CharacterType, SrcMoveData);
            if (CharacterType == CHARACTERPLAYER) {
                /* 自機のときはさらに当たり判定バッファを更新 */
                FreeAttackBuf(SrcMoveData);
                AddAttackBuf(AttackbufPlayer, TblOfs,
                  SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
                  CharacterTbl[CharacterType].Width,
                  CharacterTbl[CharacterType].Height,
                  SrcMoveData);
            }
        } else {
            /* 当たり判定バッファを更新 */
            FreeAttackBuf(SrcMoveData);
            AddAttackBuf(AttackbufMonster, TblOfs,
              SpRegTbl[SpOfs].x, SpRegTbl[SpOfs].y,
              CharacterTbl[CharacterType].Width,
              CharacterTbl[CharacterType].Height,
              SrcMoveData);
        }
    }
}

/* 指定されたテーブルのキャラクタを全部移動 */
void CharacterMoveAll(int TblLimit, TMoveDataTbl *SrcMoveTbl)
{
```

List 2E-8

```
    int i;

    for (i = 0; i < TblLimit; i++) {
        CharacterMove(i, SrcMoveTbl[i]);
    }
}

/* 敵の弾登録 */
void MonsterBulletCreate(int x, int y)
{
    CharacterCreate(MONSTERBULLETT, x, y,
                                BULLETTBLMAX, &BulletMoveTbl);
}

/* 自機の弾登録 */
void PlayerBulletCreate(void)
{
    int SpOfs;

    SpOfs = CharacterMoveTbl[CHARACTERPLAYER].SpOfs;
    CharacterCreate(PLAYERBULLETT,
      SpRegTbl[SpOfs].x + PlayerBulletAddX,
      SpRegTbl[SpOfs].y + PlayerBulletAddY,
      BULLETTBLMAX, &BulletMoveTbl);
}

/* 敵の弾移動 */
void MonsterBulletMove(void)
{
    CharacterMoveAll(BULLETTBLMAX, &BulletMoveTbl);
}

/* 自機の弾移動 */
void PlayerBulletMove(void)
{
    CharacterMoveAll(BULLETTBLMAX, &BulletMoveTbl);
}

/* ----------------------------------------------- */
/* タイマから呼び出される変数 */
void TimerEntry(void)
{
    /* 背景スクロール */
    MapScrollUp();
    /* 画面上のキャラクタ移動 */
    /* 移動とともに画面も書き換える */
    CharacterMoveAll(MOVETBLMAX, &CharacterMoveTbl);
    /* 弾移動 */
    PlayerBulletMove();
    MonsterBulletMove();
}
```

```
/* ------------------------------------------- */
/* 自機の移動と弾の発射 */

#define KEYDIRECTION_NOTMOVE      0
#define KEYDIRECTION_UPLEFT       1
#define KEYDIRECTION_UPCENTER     2
#define KEYDIRECTION_UPRIGHT      3
#define KEYDIRECTION_RIGHT        4
#define KEYDIRECTION_LEFT         5
#define KEYDIRECTION_DOWNLEFT     6
#define KEYDIRECTION_DOWNCENTER   7
#define KEYDIRECTION_DOWNRIGHT    8
#define KEYTRIGGERA               9
#define KEYTRIGGERB               10

static int PlayerMoveTblOfs;

/* 自機移動 */
/* キーは同時入力できるように、この関数を呼び出すところで     */
/* 内部のキーコード(上記の定数)にまとめておきます             */
void PlayerMove(int Key)
{
    if ((Key == KEYTRIGGERA) || (Key == KEYTRIGGERB)) {
        /* 弾の射出 */
        PlayerBulletCreate();
    } else {
        /* 自機移動 */
        CharacterMoveTbl[PlayerMoveTblOfs].MoveScriptCount = Key;
        CharacterMove(PlayerMoveTblOfs,
                      &CharacterMoveTbl[PlayerMoveTblOfs]);
        CharacterMoveTbl[PlayerMoveTblOfs].
                  MoveScriptCount = KEYDIRECTION_NOTMOVE;
    }
}
```

Section 6 カードゲーム

カードゲームはほかのコンピュータゲームと比べるとやや簡単なものと見られがちです。でも簡単だからこそ奥の深いゲームです。このカードゲームに使われるアルゴリズムとデータ構造を見てみましょう。

世界中でもっとも親しまれたゲーム

私と同年齢で，某大手メーカーの新入社員になった知人とメールを交わしているのですが，そのメールから得られた結論として「人には遊びが必要だ」ということがあります。私の場合，遊びというと「陰鬱に酒を飲む」「カラオケ」「麻雀」「映画を観る」ぐらいで，ほかには「スキー」と「旅行」があるぐらいです。こうした旅行には必ず遊び道具として携帯電話を持っていきます。旅行先から仕事が溜まっていそうな知人へ「いまお仕事してるの？　え？　私？　旅行中。おーほほほほほっ」という嫌がらせ電話をかけるのが，常に私の最重要課題となっています(笑)。

こんな「ただれた青春まっただなか」という私のことはさておき，人類史上もっとも汎用性のある遊び道具は「トランプ」ではないでしょうか？　「ポーカー」といった運を必要とするものから瞬間的な判断力が必要になる「スピード」まで，あらゆる種類のゲームがたった53枚のカードだけで楽しむことができます。

これから登場する思考型ゲームのアルゴリズムを知るには，まずカードゲームを理解しなくては先に進めません。カードゲームがコンピュータでどのようにしてゲームとして実現されているか，そのアルゴリズムとデータ構造を解き明かしていきましょう。

カードゲームとは？

何枚かの「カード」と呼ばれる紙片を使って遊ぶゲームのことを総称して「カードゲーム」といいます。先ほどのトランプが代表的ですが，トランプ以外にもさまざまな種類のカードゲームが世界中に存在します。日本には花札がかなり古い時代から存在していますし，カルタなどもカードゲームの仲間に含めてもよいでしょ

う。なおトランプという名前は本来「切り札」という意味の言葉が日本で広まったものであり，英語では「プレイングカード(Playing Cards)」といいます。

◆カードゲームで使われる用語

カードゲームにはいろいろな用語が使われます。Table 2F-1はほんの一例です。カードゲームで使われる用語は，ゲームの種類によって名称が変わることもありますが，だいたい似たような名前が使われています。

Table 2F-1●カードゲームで使われる用語

山	カードが積み上げられているところ。通常は裏返されている
場	カードを捨てたり出したりする場所
手札/持ち札	プレイヤーが持っているカード
親	ゲームを進行する際に中心となるプレイヤー
役	カードの並び方(もしくは単体)によりゲームでの勝敗や進行に関係するもの
あがり	役などが完成してゲーム終了になる
ディーラ	カードを配る人

◆カードゲームの遊び方

遊び方は多種多様です。大きく分けるとポーカーのように「運がゲームの勝敗を左右する」タイプと，コントラクトブリッジといった「ほかのプレイヤーとの駆け引きにより勝負が決まる」の2つがあります。必要なプレイヤーの数も，1人で遊ぶものから複数のプレイヤーが必要なものがあり，カードを配る枚数や勝敗の判定，役の種類など，ゲームごとに違う点をあげたら切りがありません。またカードを単体で使って遊ぶのではなく，ボードゲームやほかのテーブルゲームと組み合わせて遊ぶゲームもあります。

これだけさまざまな遊び方が生まれたのも，カードゲームがきわめて単純で，1組みのカードさえあればどこでも遊べる手軽さがあるからです。ボードゲームから端を発したロールプレイングゲームなどが「現実により近づく」という方向へ向かい，さらに複雑なシステムへと進化しているのに対し，カードゲームではすでに熟成しつくして確立した遊び方がある一方で，また違う遊び方が新しく作られているという，2つの両極端な面があります。

◆誰でも一度は遊んだことがあるはず

カードゲームはコンピュータゲームの分野でも広く楽しまれています。一部のパ

ズルゲームと同様に何度でも遊ぶことができて飽きのこないゲームが多いのもカードゲームの特徴です。Windowsに付属している「ソリティア」は，誰でも一度は熱中してプレイしたことがあると思います。カードゲームのおもしろさは「同じ展開にはならない」「ほかのゲームと比べてプレイ時間が短い」ために「手軽に何回でもトライして楽しめる」，そして何よりも「単純だからこそ熱中できる」というところにあります。

● カードゲームの中身

実際にコンピュータカードゲームを見ながら，中身を調べてましょう(Fig. 2F-1)。ここでサンプルとして取りあげるゲームは，ポーカーのように「場」と「山」があり，複数のキャラクタといっしょにプレイすることができるゲームとします。

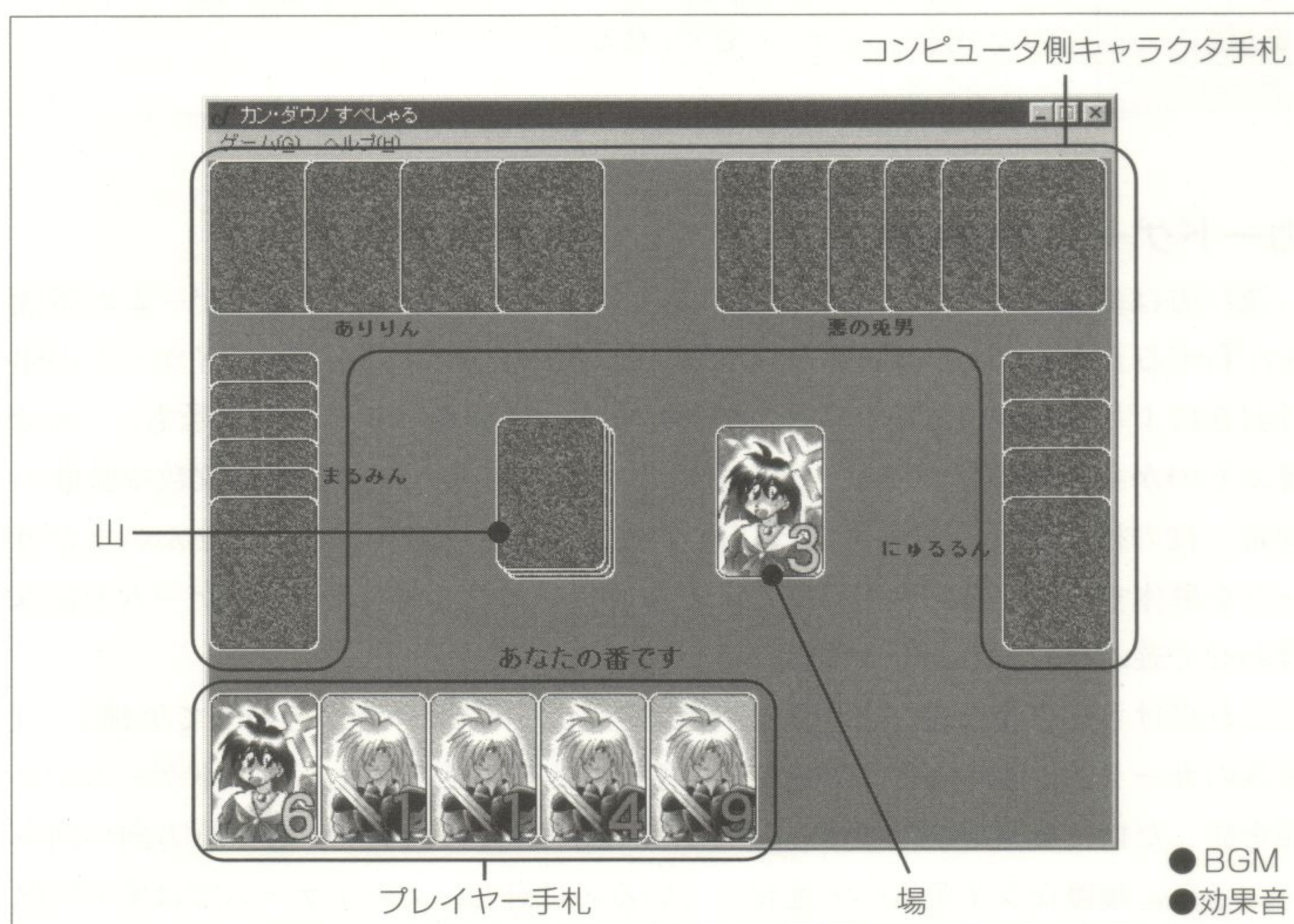



Fig. 2F-1 ●カードゲームの画面構成。ゲームプレイ時にはBGMと効果音が加わる

◆カードデータ

ゲーム画面には，「自分の手札」と「場」「山」などにカードがいくつかに分け

られて置かれているのがわかります。このときにプログラムでカードを扱うなら，カードは変数にされているはずです。カードが変数なら，プログラムはその変数を操作するだけで，カードをプレイヤーに配ったり場に捨てることができます。

◆思考ルーチン

ゲームを始めると，プレイヤー以外のキャラクタの操作はコンピュータが自動的に進めます。人間と同じようにカードを切り，ときにはプレイヤーを出し抜いて勝利することもあるでしょう。この「コンピュータが別のキャラクタとして考えてゲームを行う」手法を「思考ルーチン」などと呼びます。カードゲームでもっとも大きな比重を占めるのがこの部分です。

ゲームによってはキャラクタごとに「個性」を持たせています。人間同士でカードゲームをするとわかりますが，ゲームにはその人の性格がにじみ出てくるものです。これは思考ルーチンにキャラクタごとに優先する項目をいくつか持たせたり，キャラクタごとにまったく別のルーチンにすることで実現しています。この思考ルーチンについては後述します。

◆勝敗の判定

ゲームがある程度進むと勝敗が決まります。これを調べるのが「勝敗判定」です。手札にどんなカードがあるのかで「役が揃ったか？」「あがりか？」などを判断します。同時に得点の計算も行います。役の判断は，基本的には「総当たり」です。コンピュータはすぐには「これがどういう役なのか」はわかりません。そこでどの役になるのか，その条件をすべて手札のカードに当てはめていくことになります。この総当たりでのチェックにもいろいろ方法があります。ポーカーならロイヤルストレートフラッシュなど「判断に必要なカードの枚数が少ないもの」から判断する方法と，ワンペアなど「もっとも揃いやすい役」を優先して調べる方法があります。ゲームによっては，2つ以上の役があっても，役を組み合わせることができない場合もあるので，そのあたりも考慮して判定する順番を決めます。この判定の処理の最適化は，凝ろうとするといくらでも凝れるものです。もし判定にある程度時間がかかるようなときには，アニメーションなどを表示しながら，その裏で計算をする方法も使われています。

◆必要な３つのポイント

コンピュータのカードゲームでは，「カードデータ」「思考ルーチン」「勝敗判

定」といった3つがポイントになります。これさえあればどんな環境でもカードゲームを作ることが可能です。ロールプレイングゲームやシューティングゲームではグラフィックなどの資源をフルに活用しなければならない場面が多々ありますが，カードゲームではとくに厳しくハードウェアの処理を求めるといったことはまずありません。そのため新しいハードウェアなどで，こうした種類のゲームが比較的早く登場することになります。仕様がまだ定まらないようなマシンでも，とにかく絵が表示できさえすればゲームが作れるからです。またゲーム設定やキャラクタ作成なども比較的楽なジャンルでもあります。

● カードゲームのデータ構造

まず「カード1枚ごと」のデータを，カードの全枚数分だけ揃えなくてはいけません。1枚のカードには少なくても「数」と「種類」という2種類のデータがあります。「ハートのA」というカードなら「ハート」という種類，「A」という数の2つのデータが必要です。この2つのデータを含めた構造体を作って，それをカード枚数だけ並べたテーブルにすればほぼ完成です。

◆データの参照方法

問題はデータの参照方法です。カードデータのテーブルは「山」や「場」，プレイヤーの「手札」から相互に参照できるようにしなければなりません。実際のゲームでは，山からカードを引けばその分だけ山のカードが少なくなります。これと同じようにカードデータを操作しようとすると，メモリの削除や移動が多くなります。これは全体の管理としてはあまりよい方法ではありません。

そこで1つの変数をカードデータに含めてやります。この変数の値で「場」「山」「手札」のどこにあるカードなのかを判断するようにします。もし山からプレイヤーへカードを配るときは，この変数の値を「山」を示すものから「手札」の値に変えてから，手札のほうにこのテーブルの引数を与えます。捨てるときも同じようにします。テーブル側の値はどんなときでもそのままずっと保持しています。変わるのはこの変数の値だけです(Fig. 2F-2，List 2F-1，2F-2(P176))。同じようにしてポインタを使う方法もありますが，値が狂うとゲームのすべてが狂ってしまうので「カードを得る」関数は1つに絞り，直接にはテーブルを操作しないようにします。

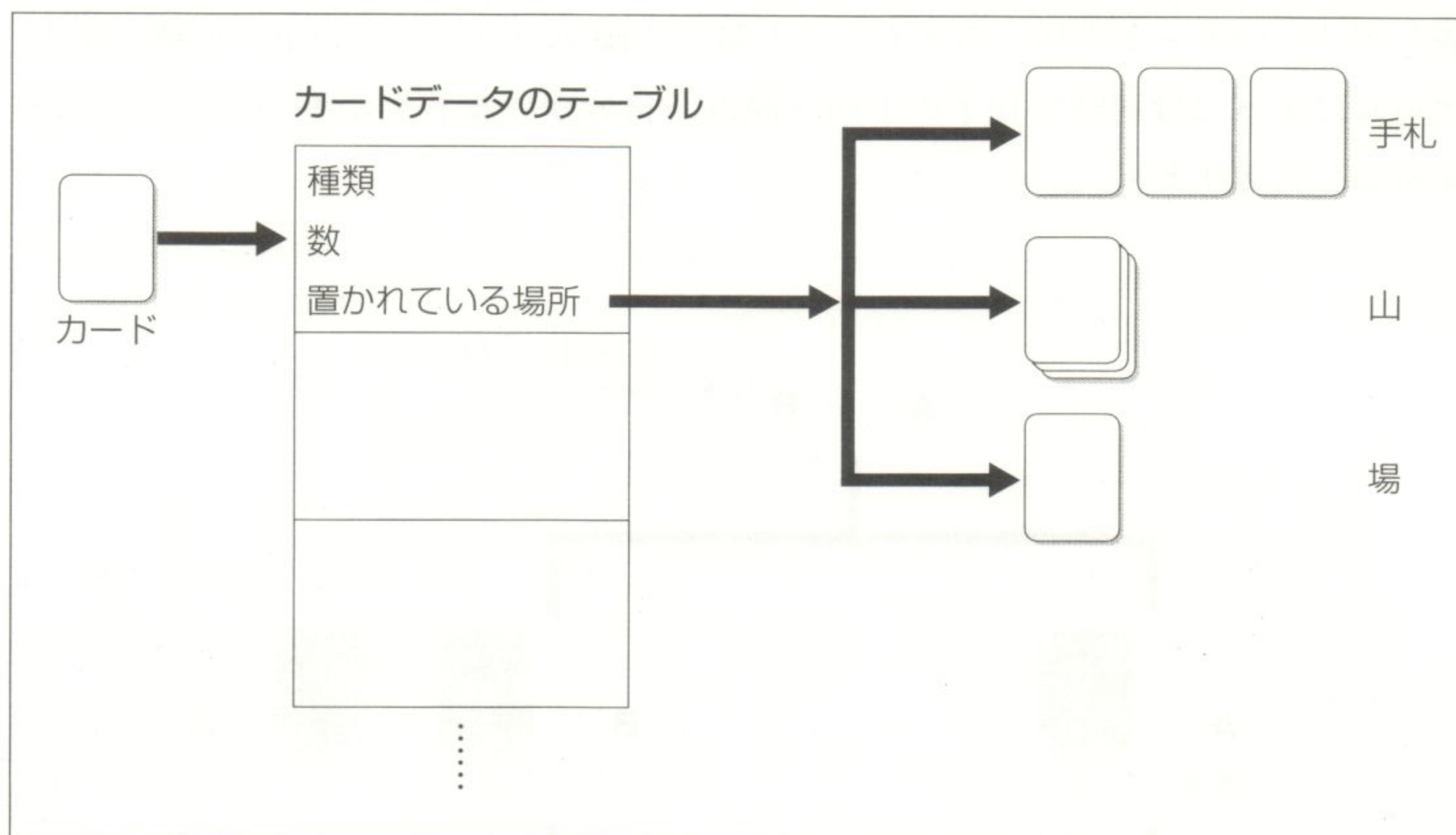



Fig. 2F-2 ●カードのデータ構造。ゲームに使うカードの枚数分のテーブルを用意し，そこに必要なデータを収める。プレイ時には，そのデータを基にしてどこにあるカードなのかを判断する

◆テーブル以外の方法

テーブルを使う以外の方法として，「場や手札に渡ったカードを記憶」しておき，新しくカードが必要になったときは「いままで出た以外のカードを新しく作り出す」という方法もあります。

まず「カード」を得る関数を作ります。カードに必要なパラメータのメモリを確保し，種類などを乱数で生み出します。このときに，この種類が過去のカードと重複しないかをチェックする処理が必要です。そしてカードを場や手札として渡すときに，それをリストの1つとして登録します。これは動的なリストで，生み出した順にポインタで結び付けていくことになります。ゲーム中にすべてのカードをめくるのでなければ，占有するメモリが比較的少なくて済むのがこの方法の利点です。

● カードゲームの思考ルーチン

思考ルーチンとはいったいどんなものでしょうか？　すぐに頭に浮かぶものとしては「先の手を読んでいき，もっとも勝てる確率が高いものを選ぶ」方法でしょう(Fig. 2F-3)。たとえば捨てるカードを選ぶときは，場に出たカードなどから判

断して「どのカードを捨てるともっとも勝てる確率が高くなるのか」を調べます。ゲームによっては同時に相手の手筋を読み，自分がいま有利か不利なのかを判断することもあります。

A B どちらを捨てるか？
というときには
A Bを捨てた場合
B Aを捨てた場合
は確率的に
きやすいカード
A 新しくきたカード
を捨てた場合
Aを捨てた場合
B 新しくきたカード
を捨てた場合
Bを捨てた場合

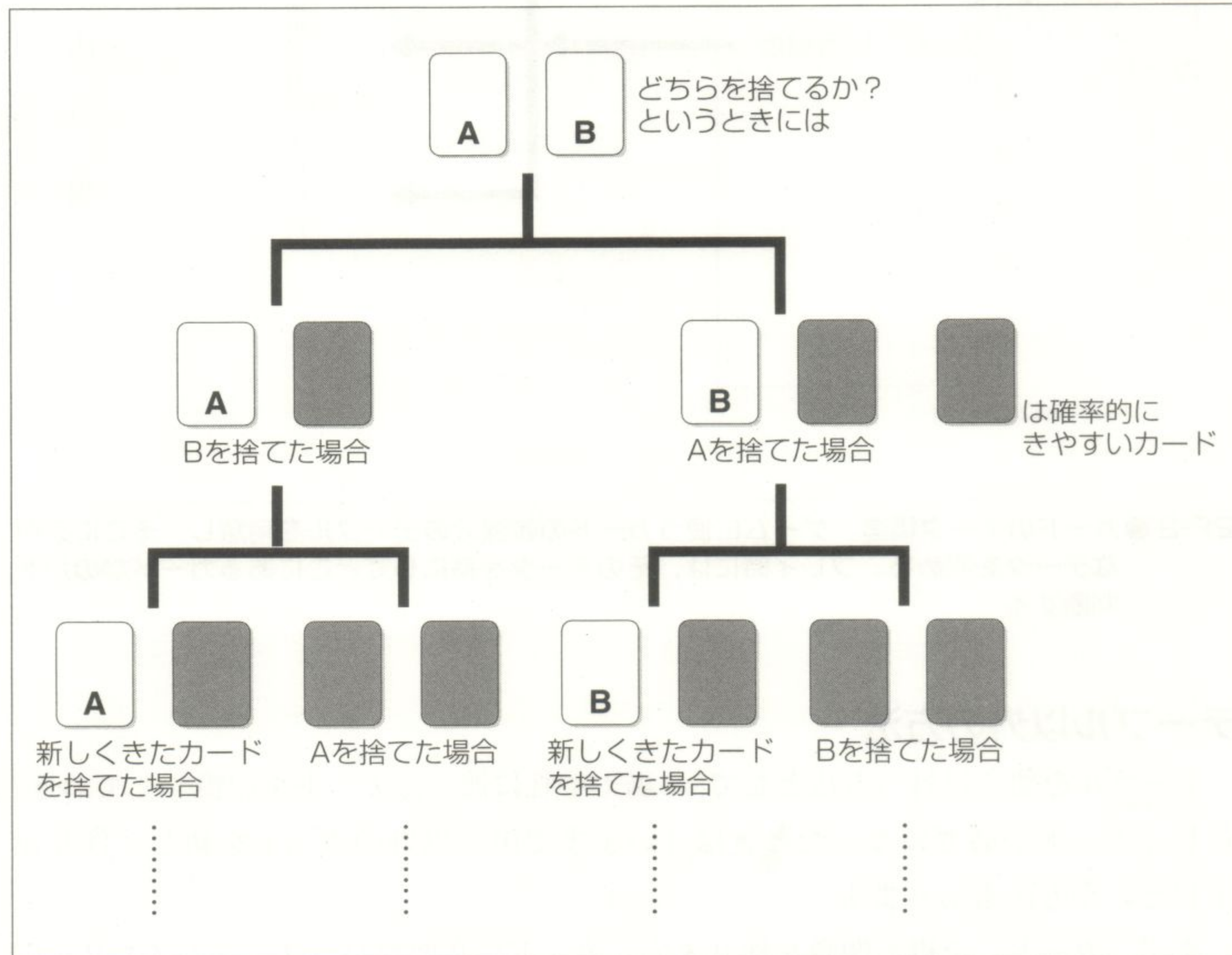



Fig. 2F-3 ●先の手を読んでいくタイプの思考ルーチン

◆さまざまな思考ルーチン

ポーカーなど運に頼るゲームでは上記の方法はやや不向きですが，そのときには視点を変えて「どのカードがもっとも引きやすいか」を予測したうえで「どの役が作りやすいか」ということを目的にします。Fig. 2F-4はその具体的な例です。

この方法の問題点は「確率を調べるのにかかる計算時間」です。単純に総当たりで調べると，残っているカードや条件が多いほど予想される組み合わせは指数的に増えていきます。そこで計算時間を短くするため，さまざまなアルゴリズムが開発されました。よく使われるのが「見込みがなさそうな組み合わせは切り捨てていく」という方法です。調べている途中で前に調べた組み合わせのほうが確率が高いということがわかれば，そのときは，いま調べている組み合わせを放棄し

て，別の組み合わせを調べるようにします。さらに「前回調査した見込みのない組み合わせは最初から削除する」「もっとも確率が高そうなものから優先して調査する」などを組み合わせることでより計算速度が上がります。

「先の手を読む」以外にも方法はあります。実際にあなたがカードゲームをプレイするときの状況を思い浮かべてください。カードを捨てるときに「先に絵札から捨てたほうが安全」「最後まで役札を取っておけば勝てる」といったことを考えるはずです。実はどんなゲームにも必勝法というものがあります。これをルーチンとして条件式にまとめればいいのです(Fig. 2F-5)。また前述の確率を調べる方法にこの必勝法を組み合わせると，さらに強い思考ルーチンを作ることができます。

思考ルーチンは立派な学問の一分野として確立しているほどで，前述の方法はあくまで基本的なものにすぎません。より深く知りたくなったら，人工知能関連の書籍を探してみてください。いろいろな方法が研究されているのがわかると思います。

どの役が作りやすいかを調べる
確率が低ければもう一度
その役に必要なカードが確率的にきやすいか調べる
場，手札から残っているカードを調べる
確率が高ければその役に不要なカードを捨てる
捨てるカード

Fig. 2F-4 ●ポーカーの思考ルーチン

2 9 0 7 5

同じ種類のカードはあるか？
同じ数のカードはあるか？
高い役を作れるカードはあるか？
損になるカードはあるか？

必勝法を条件式にする

2

捨てるカード

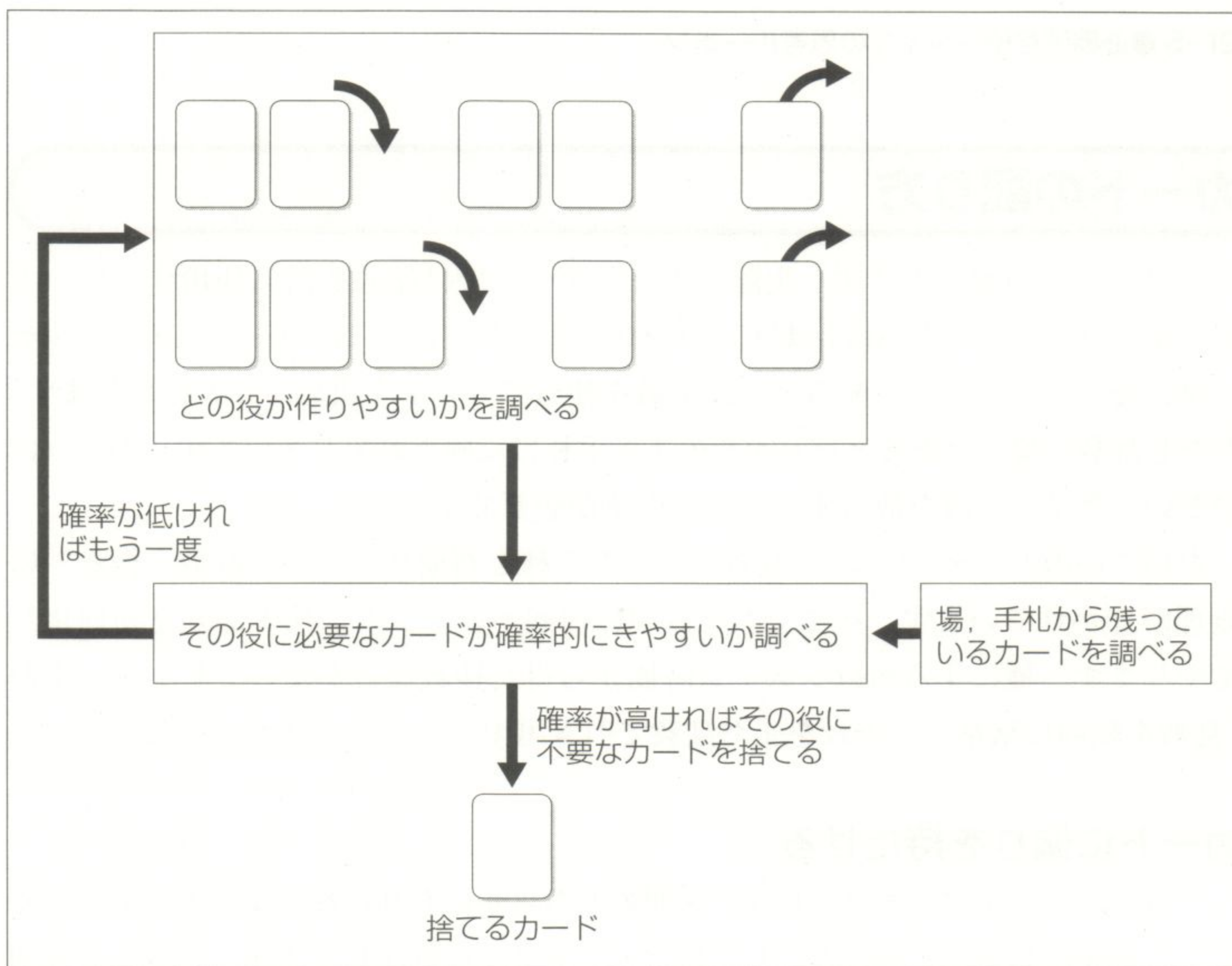
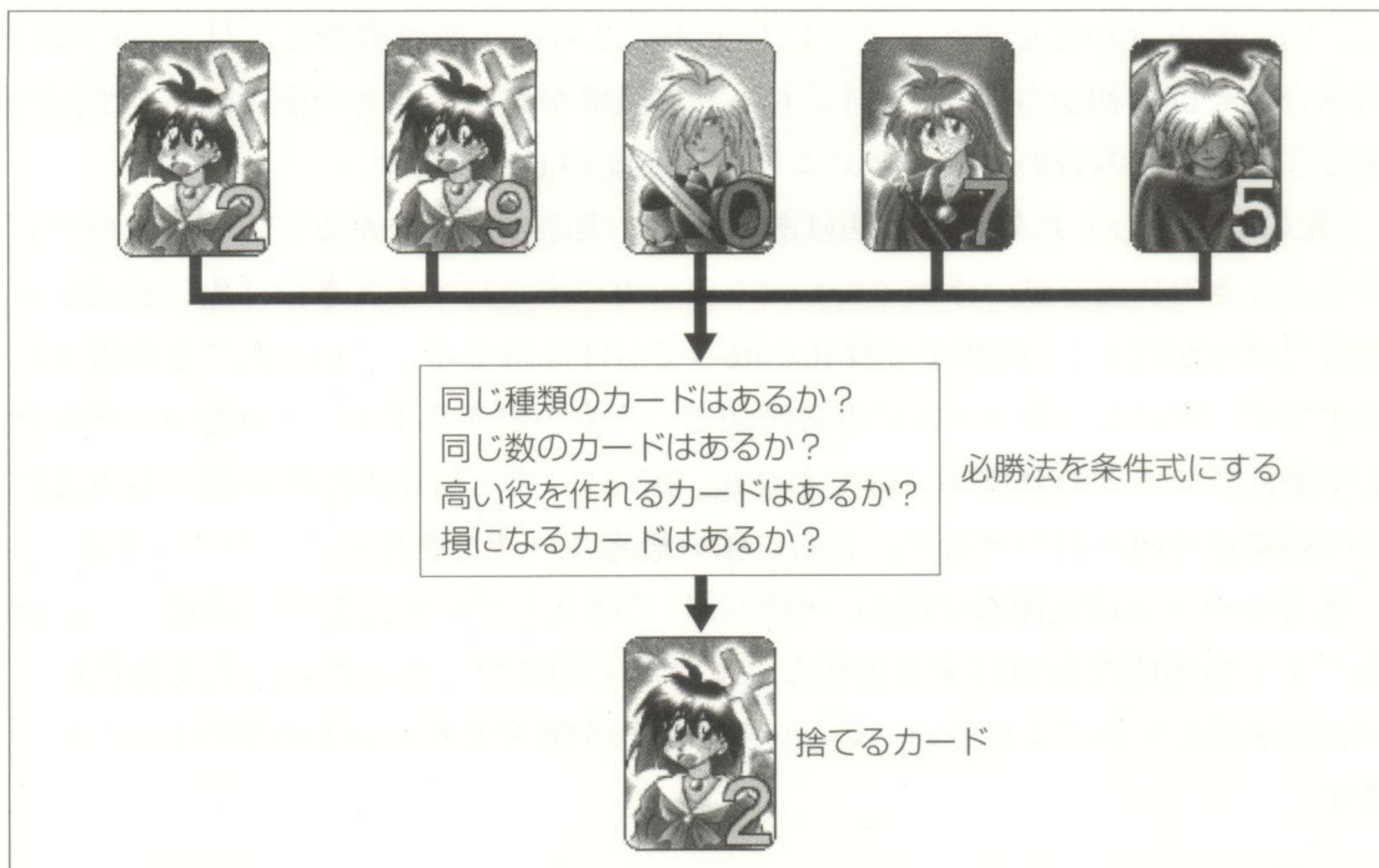



Fig. 2F-5●必勝法を使うタイプの思考ルーチン

カードの配り方

次にカードの配り方です。実際のカードゲームでは始める前に使用するカードをシャッフルしてから山にします。コンピュータゲームでは，カードはテーブルデータになっているので，配るときに乱数を使ってカードを決めることになります。ただし乱数に偏りがあるとそれがそのまま手札に反映されてしまいます。あまりにひどいときは「乱数を散らす」という作業が必要です。

たいていのシステムでは，「乱数の元となる種を初期化する」，「乱数を得る」に処理が分かれています。そこでゲーム中に何回かランダムに種の部分を初期化してやります。種にする値はシステム時間から得た秒数などがよいでしょう。また「乱数を配列に格納してそれをまた乱数で引き出す」という方法も使われます。

◆カードに偏りを持たせる

一方で，プレイヤーかコンピュータ側のどちらかに有利になるようカードを配るゲームが存在します。ゲームとしてはフェアではない気がしますが，こうした有利不利をランダムや条件次第で切り換えているゲームは意外と多くあります。たとえば「アイテムを使う」「あるキャラクタに勝ち続ける」というときに，妙によい

カードが配られるといったゲームです。方法としては，有利なカードというのはゲームルールによって固定されていたり手札に左右されるので，まずはそのカードを探すところから始まります。役を作るゲームでは，手札からどの役が完成しやすいかを調べて，その役に足りないカードを配ることになります。ポーカーなら最初からツーペアぐらいの役が手札に入れられたりします。

さらにひどいゲームとなると，コンピュータ側のキャラクタにはあらかじめ高い役が入っていて，ゲームがある程度進んだところでそれを使ってあがるということがあります。これはゲームではありません。ゲームを作っている側としてもプレイヤーに遊んでもらってもつまらないものです。このような形ではなく，プレイヤーもコンピュータ側も同じ条件で，あくまでゲームの流れとして，有利なカードを配るようにする方法がよいと思います。

◆カードを配る関数を用意する

カードには，「カードを配る関数」を用意しておきます(List 2F-3，2F-4(P178))。これ以外の関数からカードを取るようなことはしないようにします。こうしておかないと，のちのちデータ構造を変更したときに，非常にめんどうなことになってしまいます。またバグも発生しやすくなります。

● カードゲームの流れ

以上のアルゴリズムを使って，全体的なゲームの流れを形にしてみましょう(Fig. 2F-6)。ゲームを開始するときは「プレイヤーの順番を決める」「親決め」「最初のカードを配る」といった処理をします。ゲームの開始はここからです。プレイヤーの番になったらカードを切ることになります。キー操作などによりカードが指定されたら，そのカードを場に捨てる処理をします。次はコンピュータ側のキャラクタの番です。前述の思考ルーチンを使ってカードを切ります。これが繰り返されていき，誰かがあがりとなったらゲーム終了です。得点を計算したり次のゲームへの準備をします。カードゲームはこの繰り返しで進められていきます。

● サンプルゲームの遊び方

ただのトランプゲームではありきたりでおもしろくないので，「ババのないババぬき」という感じのゲームにしてみました。エンディングはありませんが，何度で

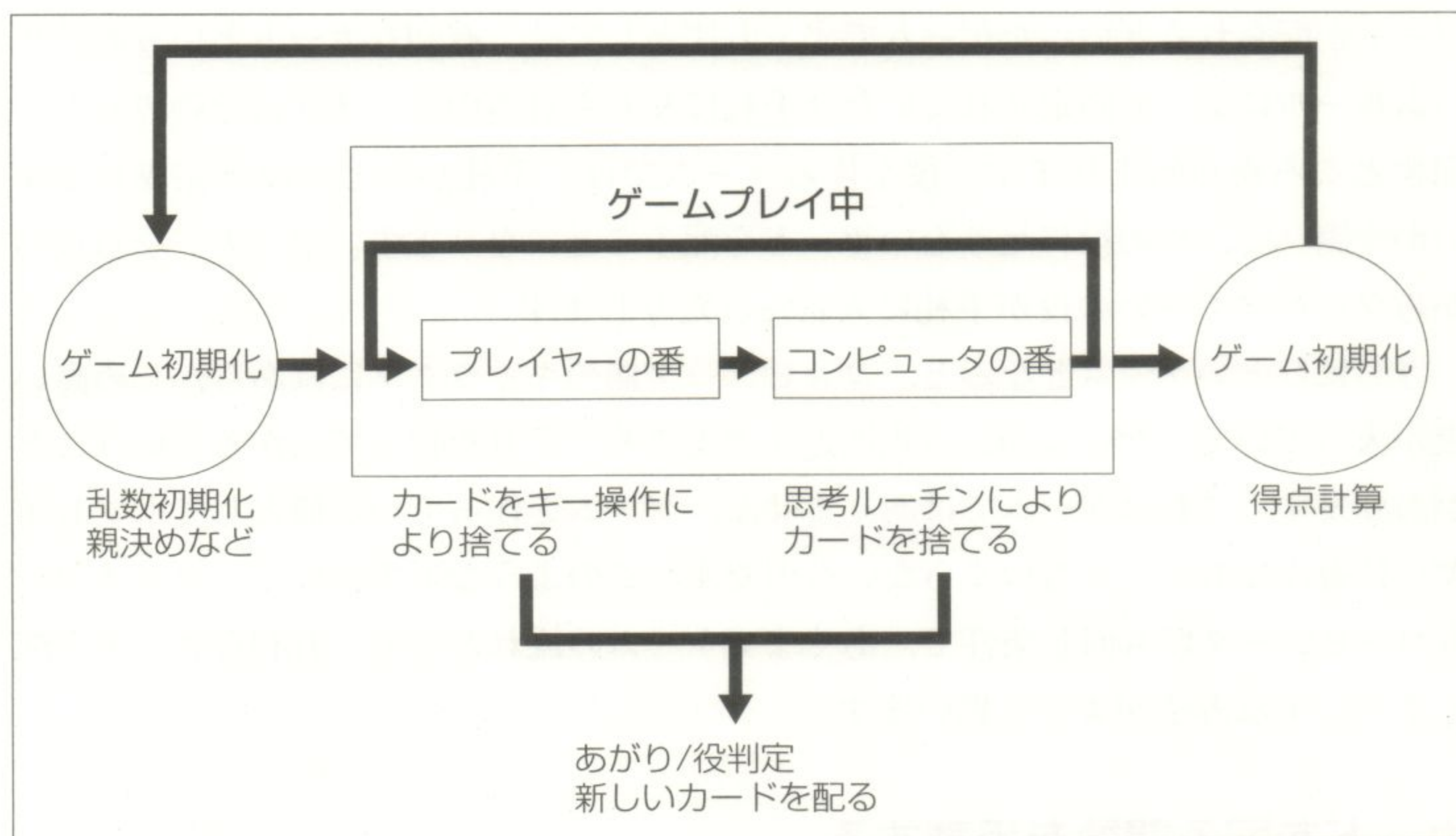



Fig. 2F-6 ●カードゲーム全体の流れ

も楽しめるカードゲームにしたつもりです。

コンピュータ側キャラクタの思考ルーチンは「必勝法」による方法にしています。カードの表示にはシューティングゲームなどで使ったスプライトを流用しました。あがりの条件が「手札がなくなる」ことなので役の判定は入っていません。そのかわり出したカードによって役があるので，そのあたりのルーチンを加えています。

意外と簡単な作りなのですが，作った本人がいちばん勝てません(笑)。イラスト担当のMALUMI氏とともに，この本で作ったゲームのなかではよく遊んでいるゲームです。ぜひ遊んでみてください。

List 2F-1 ●カードのデータ構造（Delphi）

```
{ 定数宣言 }
const
    CARDTBLMAX      = 53;
    POSITIONPOOL    = 0;  { カードは山にある  }
    POSITIONHAND    = 1;  { カードは手札にある }
    POSITIONFIELD   = 2;  { カードは場にある  }

{ 型宣言 }
type
    TCardData = record          { TCardData 構造体の定義 }
        Category: integer;      { 種類                  }
        Number: integer;        { 数                    }
        Position: integer;      { 置かれている場所      }
```

```
    end;
    TCardDataTbl = array[0..CARDTBLMAX] of TCardData;
    pTCardDataTbl = ^TCardDataTbl;


{ 変数宣言 }
var
    { カードデータのテーブル }
    CardDataTbl: array[0..CARDTBLMAX] of TCardData;
```

List 2F-2 ●カードのデータ構造(C/C++)

```
/* 定数宣言 */
#define CARDTBLMAX       53
#define POSITIONPOOL     0    /* カードは山にある    */
#define POSITIONHAND     1    /* カードは手札にある */
#define POSITIONFIELD    2    /* カードは場にある    */

/* 型宣言 */
typedef struct {           /* TCardData 構造体の定義    */
    int Category;          /* 種類                      */
    int Number;            /* 数                        */
    int Position;          /* 置かれている場所          */
} TCardData;

typedef TCardData TCardDataTbl[CARDTBLMAX + 1];
typedef TCardDataTbl *pTCardDataTbl;

/* 変数宣言 */
/* カードデータのテーブル */
TCardData CardDataTbl[CARDTBLMAX + 1];
```

List 2F-3 ●カードを配る関数(Delphi)

```
(* CardListの引数をランダムに得る *)
function GetRandomCardIndex: integer;
var
  TmpIndex, Index: integer;
begin
    Index := Random(CardLimit);
    (* 前方検索 *)
    for TmpIndex := Index to CardLimit do begin
        if CardDataList[TmpIndex].Position = PositionPool then begin
            result := TmpIndex;
            exit;
```

List 2F-3

```
        end;
    end;
    (* 後方検索 *)
    for TmpIndex := Index downto 0 do begin
        if CardDataList[TmpIndex].Position = PositionPool then begin
            result := TmpIndex;
            exit;
        end;
    end;
    (* 全部場に出た *)
    result := -1;
end;

(* 山から1枚引く *)
function GetRandomCard(Player: integer): integer;
var
  Index, TmpIndex: integer;
begin
    { 空いているカードを得る }
    Index := GetRandomCardIndex;
    { カードの空きがなかった }
    if Index = -1 then begin
        { なかったら場に出たものをもういちど山に戻す }
        for TmpIndex := 0 to CardLimit do begin
            if (CardDataList[TmpIndex].Position = PositionField) and
              (TmpIndex <> FieldTop.ListIndex) then
                CardDataList[TmpIndex].Position := PositionPool;
        end;
        Index := GetRandomCardIndex;
        if Index = -1 then begin
            result := -1;
            exit;
        end;
    end;
    { カードデータに状態を設定 }
    CardDataList[Index].Position := PositionHand;
    CardDataList[Index].Player := Player;
    { 得たカードを返す }
    result := index;
end;
```

List 2F-4●カードを配る関数(C/C++)

```
/* CardListの引数をランダムに得る */
int GetRandomCardIndex(void)
{
    int TmpIndex, Index;

    Index = Random(CardLimit);
    // 前方検索
    for (TmpIndex = Index; TmpIndex < CardLimit; TmpIndex++) {
        if (CardDataList[TmpIndex].Position == PositionPool)
            return TmpIndex;
    }
    // 後方検索
    for (TmpIndex = Index; TmpIndex >= 0; TmpIndex--) {
        if (CardDataList[TmpIndex].Position == PositionPool)
            return TmpIndex;
    }
    // 全部場に出た
    return -1;
}

/* 山から1枚引く */
int GetRandomCard(int Player)
{
    int Index, TmpIndex;

    // 空いているカードを得る
    Index = GetRandomCardIndex();
    // カードの空きがなかった
    if (Index == -1) {
        // なかったら場に出たものをもういちど山に戻す
        for (TmpIndex = 0; TmpIndex < CardLimit; TmpIndex++) {
            if ((CardDataList[TmpIndex].Position == PositionField) &&
              (TmpIndex != FieldTop.ListIndex))
                CardDataList[TmpIndex].Position = PositionPool;
        }
        Index = GetRandomCardIndex();
        if (Index == -1)
            return -1;
    }
    // カードデータに状態を設定
    CardDataList[Index].Position = PositionHand;
    CardDataList[Index].Player = Player;
    // 得たカードを返す
    return Index;
}
```

Section 7 麻雀ゲーム

麻雀は非常に楽しいゲームですが，コンピュータゲームとして作るにはややめんどうなものでもあります。このコンピュータ麻雀ゲームから思考ルーチンの組み立て方法や役の判定などを中心にお伝えします。

● 今宵あなたと一晩中……

私にとっての重要な遊びの1つに「麻雀」があります。所沢にいる知り合いの家へ仲間同士が集まって徹夜で麻雀をするのが，ほぼ毎月の行事になっています。

麻雀のおもしろさはいったいどこにあるのでしょうか？　いい歳したおとなが集まって，一晩中タバコの煙にまみれながら麻雀という底なしのゲームを何度も繰り返しています。しかも時間が過ぎていくと少しずつナチュラルハイの状態へと近づき，狂気の香り漂う何ともいえない独特の雰囲気が生まれてきます。はた目から見ればちょっと不気味です。それでも楽しいものは楽しく，飽きもせずいつも集まっています。

この不思議なおもしろさが隠されている麻雀の本質を探りながら，コンピュータ麻雀ゲームのアルゴリズムに触れていきます。そうすることでコンピュータの思考ルーチンというものがわかるはずです。

● 麻雀とは？

中国に端を発したこのゲームは，アジア系民族を中心に広く楽しまれています。とくに香港では結婚式の披露宴などのお祝いの席に欠かせません。日本ではギャンブルの対象としてやや暗いイメージがあるようですが，庶民的なゲームとして多くの人に親しまれています。

◆麻雀のルール

ゲームルールはカードゲームなどと比べるとやや複雑なものとなっています。覚えなければならないルールや役が多くあり，これが勝敗に密接にかかわるため，覚えたてのころは何度も負けてしまいます。でも，これを通り越すとだんだんとおも

しろくなってきて，病みつきになるようです。私の場合はこの本でグラフィックを担当しているMALUMI氏の「振らなきゃ勝てる」という言葉で「守り」を知ってから強くなりました。

ローカルルールが非常に多いのも特徴です。とくに役の種類や扱い方が，場所や人によって違います。たとえば，ドラ関係では5筒を赤くした「赤ウー」などを混ぜてそれをドラと同じ意味にすることもありますし，ドラ表示牌は普通は1枚だけめくりますがこれを2枚にするところもあるようです。ときにはこうしたルールの違いがトラブルの元になりますが，それもおもしろさの1つにもなっています。

◆コンピュータゲームとしての麻雀

コンピュータによる麻雀ゲームといえば，「脱衣麻雀(脱ぎ麻雀)」がもっとも有名かもしれません。コンピュータ側のキャラクタと麻雀で闘い，プレイヤーが勝つと対戦した相手が少しずつ服を脱いでいき，そのグラフィックを見ることができるというものです。この脱衣麻雀で麻雀を覚えたという人も多いと思います。種類も豊富で，アーケードマシンからコンシューママシンまで幅広く浸透しています。最近では，著名なイラストレータがキャラクタデザインやグラフィックを担当したり，ただ脱ぐだけではなくアドベンチャーゲームのようなストーリ性を持たせたものが主流になっています。

もちろん本格的な麻雀ゲームもあります。麻雀の「4人いなくてはゲームができない」という点を，コンピュータ側がほかの3人を代行することで「いつでも好きなときにゲームができる」手軽な遊びとなるのです。さらにネットワークに対応した麻雀ゲームでは，離れたところにいる人と対戦することもできます。また麻雀牌を利用したパズルゲームなどにも発達しました。これらも根強い人気があります。

コンピュータゲームとしての麻雀では，実際の麻雀と牌の表示方法やルールがやや変わっています。先ほどの脱衣麻雀では4人でプレイするのではなく，1対1で闘うゲームに変形されていることが多いようです。また4人制のゲームでも，卓を囲む形ではなく，捨て牌が横へ4列に並んだ形で表示されるものもあります。いずれも実際に卓を囲むような表示にすると，牌の表示などが小さくなったりして，ムダの多いレイアウトになってしまうので，こうした工夫がされています。

◆人間臭さこそが醍醐味

麻雀はカードゲームと同じく，あまり厳しいハードウェア環境や処理は必要としません。ただ使われる思考ルーチンや役の判定方法がややめんどうなので，最

初から作るには荷が重いゲームです。そのため，一度作られたアルゴリズムが，えんえんとそのゲームメーカーで使われ続けることもよくあります。

麻雀の楽しいところはギャンブル性は差し引くとしても「人対人」で闘う点にあります。ほかのギャンブルでは勝つために「運」の要素がとても大きく働きますが，麻雀ではこれに対戦しているプレイヤーとの「駆け引き」と「戦略」がかかわってきます。ゲーム進行や役の作り方は人によって大きく異なります。そこにプレイヤーの人間性が強く出てくるのです。また大きな役を作っている最中に，先に相手にあがられたときの悔しさというのも人間らしい感情といえます。ほかのゲームではあまり感じられないこうした「人間臭さ」こそ麻雀ゲームの醍醐味です。コンピュータゲームでは，そのすべてを実現することができません。そのため，コンピュータの麻雀ゲームでは，「麻雀そのものを実現する」のではなく「麻雀のいくつかの要素をバランスよく再現する」ことになります。

麻雀ゲームの中身

4人打ち方式の麻雀ゲームからゲームの中身を探ってみましょう。

画面には実際の麻雀と同じように卓や牌を表示しています(Fig. 2G-1)。カードゲームのカードと同じく，牌は変数として扱われています。山から牌をツモときは，牌と対応する変数の値が山から手牌へと操作されているだけです。

コンピュータ側キャラクタ手牌
山
ドラ牌表示
ゲーム進行
得点
捨て牌
プレイヤー手牌
●BGM
●効果音

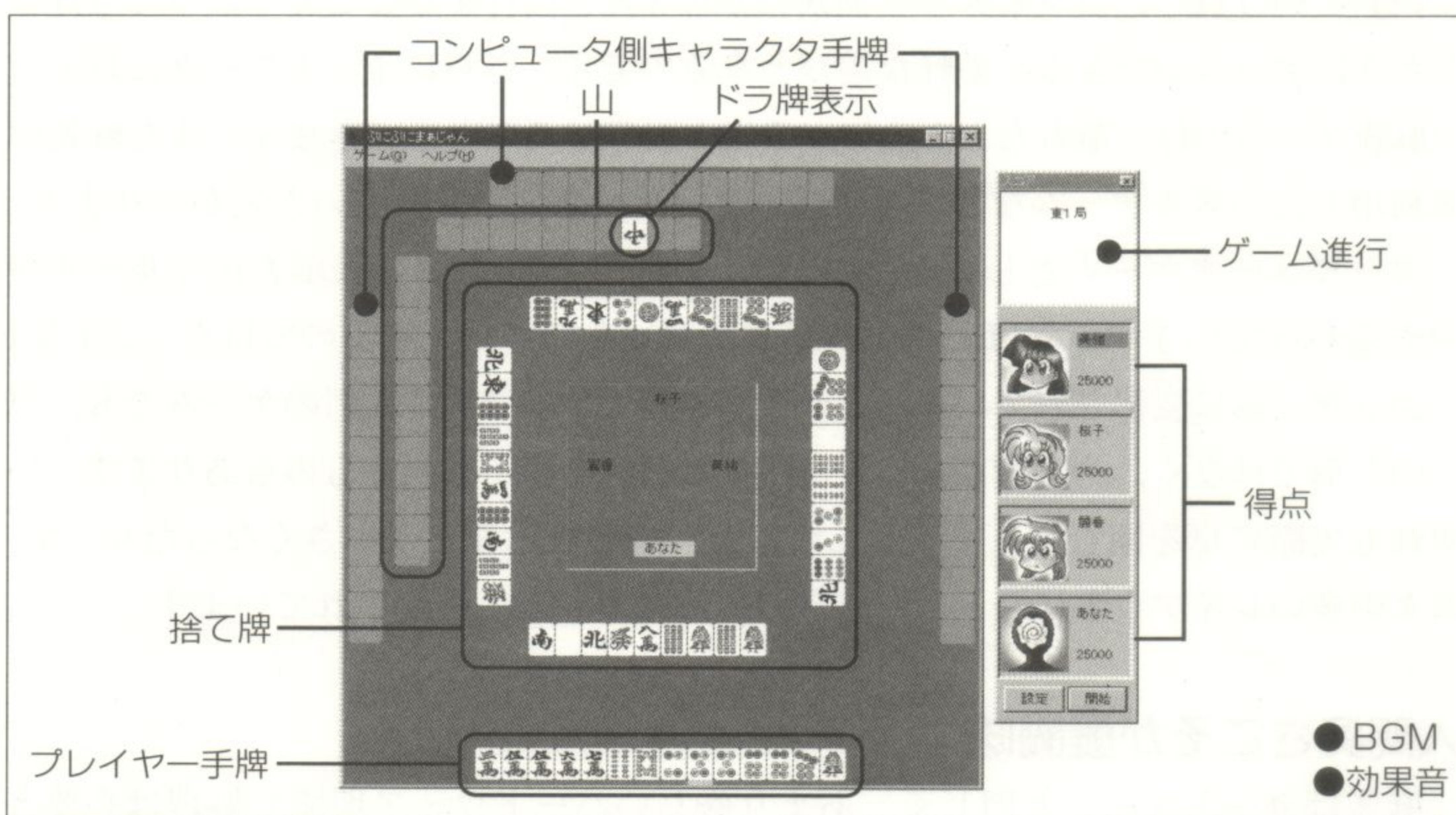



Fig. 2G-1 ●麻雀ゲームの画面構成。ゲームプレイ時にはBGMと効果音が加わる

◆思考ルーチン

プレイヤー以外の対戦相手は，コンピュータ側で処理されます。これもカードゲームと同じく「思考ルーチン」を使っています。ほとんどの麻雀ゲームでは，対戦相手のキャラクタによって「強い」「弱い」などいくつかの性格が設定されています。麻雀ゲームでは思考ルーチンがめんどうなものになりやすく性格が出しにくいため，配牌の段階で有利なものがくるようにしてキャラクタごとの強弱を付けるようにしているゲームが多数あります。

◆ゲームならでは表現方法

実際にゲームをプレイしてみると「牌を扱うときの動作」が現実の麻雀と違うのがわかります。リーチのときは「リーチ！」と叫ぶかわりに，それを指示するボタンなどを押してから牌を切ります。「ポン」「チー」「カン」などは場に出た牌によって選択肢が書かれたメニューが表示されます。これは牌が捨てられたときにプレイヤーの手牌をチェックして，必要な牌かどうかを調べています。

リーチを宣言すると，あとはコンピュータが自動でツモと捨て牌を行います。あたり牌もコンピュータが知らせてくれます。役の判定や得点計算，点棒配分など，全部コンピュータ任せです。このあたりがコンピュータ麻雀の楽なところですが，プログラマにとっては作るのにたいへん労力が必要な部分でもあります。

◆麻雀ゲームのポイント

麻雀ゲームそのもののポイントとしては，この「役の判定」「得点計算」と「牌の取り扱い」「思考ルーチン」あたりが重要なものとなっています。とくに「役の判定」はあがったとき以外にも「あたり牌の判定」「思考ルーチンでの捨て牌選択」といったものにも使われます。まずこの重要な4つの部分について，アルゴリズムを詳しく見ることにしましょう。

● 麻雀牌のデータ構造

麻雀牌のデータ構造は，基本的にカードゲームと同じです。1つの牌に含まれる情報をそのまま構造体としてまとめたら，それを牌の数だけテーブルにします(Fig. 2G-2, List 2G-1, 2G-2(P192))。ただし，あとで役の判定を簡単にするために，「1か9の牌(老頭牌)」「それ以外の数の牌(中張牌)」「字牌」といった情報も持たせるようにします。たとえば「9萬」なら「万子」「9」「老頭牌」という感じです。

牌データテーブル

牌

種類（万子など）

属性（公九牌，字牌など）

数（1，2，3……）

置かれている場所

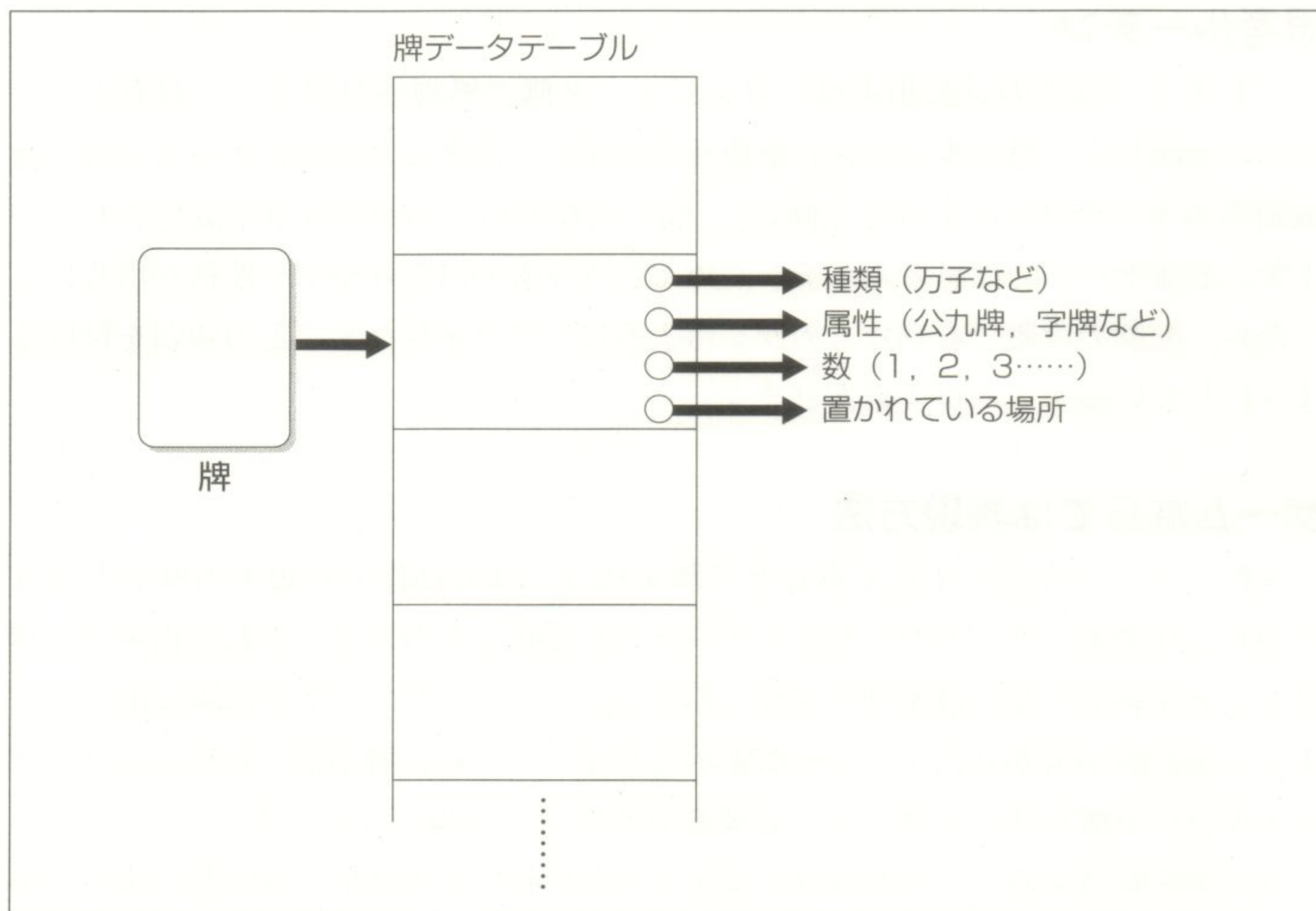



Fig. 2G-2 ●牌のデータ構造。すべての牌の数だけテーブルを用意し，そこに牌1枚分ずつ必要なデータを収める

字牌では「東/南/西/北……」などのように並び順が決まっているので，それぞれ頭から「0/1/2/3……」というように数のところに対応する値を入れます。

◆位置を示すフラグを入れる

また牌がどこにあるかを示すフラグも加えておきます。「場」「手牌」「山」といったもののほかに，「ドラ表示牌」「ないた牌」などもフラグに含めます。このあたりの処理はカードゲームと同じです。

麻雀では，牌はばらばらに混ぜたあとに山として積みますが，コンピュータゲームでは「牌を得るときに乱数で取り出す」ようにすると，よけいな配列や作業が省けて便利です。牌を取り出すときは乱数で牌のテーブルを引きます。引き出したテーブルからデータを読み出して「山」にフラグがあるものを使うようにします。もし乱数で選ばれたテーブルが山以外のものならなら，テーブルの前方か後方に向けてフラグが山にある牌を探します。この検索方向の切り換えも乱数を使ってランダムにするとよいでしょう。こうした処理は「牌を得る」関数の中で行い，ほかのルーチンからはこの関数を使って参照するようにします。

役判定のアルゴリズム

役の判定を行うアルゴリズムを考えてみます(Fig. 2G-3, 2G-4)。

麻雀の手牌は基本型として「3つの牌」が4組と，頭と呼ばれる「同じ牌が2つ」1組の組み合わせが完成したときにあがり役となります。

Fig. 2G-3●牌のチェックの流れ

国士無双のチェック
○風牌が4または5枚
○三元牌が3または4枚
○そのほかの牌が6または7枚

七対子のチェック
○すべての牌が2枚ずつ
○違う種類2つで6枚なら
シュンツがあるかどうかチェック
シュンツ4組なら二盃口

ピンフのチェック
○頭以外すべてシュンツ

トイトイ系のチェック
頭以外コーツ/カンツですべてできている
Yes
暗刻はいくつあるか?
3個 → 三暗刻
4個 → 四暗刻
槓子はいくつあるか?
3個 → 四槓子
4個 → 三槓子

タンヤオのチェック
○すべての牌が中張牌

チャンタ系のチェック
すべての牌が公九牌 —Yes→ 清老頭
No
すべての組み合わせで公九牌が含まれる —Yes→ ジュンチャン
No
字牌以外の牌に公九牌が含まれる —Yes→ チャンタ

チンイツ系のチェック
すべての牌が同じ種類 —Yes→ チンイツ
チンイツ → 字牌か?
字牌か? —Yes→ 大四喜
字牌か? —No→ 九蓮宝燈のチェック
No
字牌以外の牌が同じ種類
Yes
ホンイツ

風牌のチェック
三元牌が3枚ある —Yes→ 大三元のチェック 小三元のチェック
No
風牌が4枚ある —Yes→ 大四喜のチェック 小四喜のチェック

Fig. 2G-4 ●役の判定を行うアルゴリズム

◆手牌の数と種類を調べる

最初の段階として，牌の種類ごとに数が「3の倍数」か「3の倍数＋2」または「2」なのかを調べます。

もし全種類の牌が「2の倍数」のときは「七対子」かどうかを調べます。同じように風牌が「4または5」三元牌が「3または4」あるときは「国士無双」を疑ってみます。こうして最初の段階で「七対子」「国士無双」「それ以外の役」の3つに分けます。

①牌の種類がいくつあるか調べる
種類ごとの枚数から
「3の倍数」
「3の倍数＋2」
「2の倍数」
のいずれかを結果として得る

②種類の倍数からある役を判断
すべての牌が「2の倍数」だった
七対子→2枚ずつ取り出し差がすべて0になるもの
なおかつ二盃口ではないもの
国士無双→風牌が「4または5」，三元牌が「3または4」
それ以外
ほかの役としてチェック

③配の並び方が完成しているか調べる
2枚ずつチェックする
1．最初のほうから2枚の牌を取り出す
2．牌が数の多いほうから低いほうへ引いて
「1」だったら3へ
「0」なら5へ
「2」ならへカンチャン待ち
3．もう1枚次の牌を持ってきて，前の牌の大きいほうと引く
また「1」なら「シュンツ」
4．上で「1」以外なら「ペンチャン待ち」を調べる
数の大きいほうが9，または小さいほうが1ならそう
5．もう1枚次の牌を持ってきて，前の牌の大きいほうと引く
また「0」なら「コーツ」
6．上で「0」以外なら「トイツ(頭)」
次に牌を取るときは引いてきた牌からにする
7．これを最後まで繰り返す
この段階で「シュンツ」「トイツ」「カンツ」の数を出しておいて役の判断に利用する

Fig. 2G-5●手配の調べ方

次の段階では，牌に書かれている数そのものよりも「数の並び方」が問題となります。先ほどの「3つ牌」の組み合わせでは，9－8－7のように同種の牌の数が順番に並んでいる「シュンツ(順子)」と，同じ牌が3枚ある「コーツ(刻子)」があります。「同じ牌が2つ」の場合は「トイツ(対子)」と呼ばれています。これは「頭」ともいって，あがり役に必ず1つ必要です。これらを調べることで「あがり役がきちんと完成しているかどうか」「待ち牌の形」「頭の種類」などを得ることができます。調べ方は(Fig. 2G-5)のとおりです。

◆役を判定する

あとは役そのものの判定です。麻雀の役には「ある役が完成しているときは，別の特定の役ができていても得点に加算されない」といった排他的な役が多く設けられています。またメンゼンでなければ役として成立しないものもあります。まず先にそうした条件を調べます。

役満では基本的にほかの役は加算されません。役満の種類をざっと見ると「清一色の形になっている(緑一色，字一色，九連宝燈)」ものと「三暗刻から発展したもの(四暗刻，大三元，四喜和など)」，そのほかに「すべて老頭牌(清老頭)」などがあります。そこで，この段階で上記の条件に合う形になっているのであれば，役満のチェックをします。

テンパイの判断はこの役判定ルーチンを応用するだけです。「数の並び」をチェックする段階で，「シュンツには1つ足りない」または「トイツが2つある」などが検出できます。またいっしょに待ち牌の状態も調べられます。

役の判定方法には意外と多くのバリエーションがあります。この例はもっとも素直な方法です。これ以外には「牌の分布を調べる」「いくつかの牌にかぎってテーブル引きする」などの方法があります。基本的なソートと統計を調べる問題でもあるので，いろいろ最適化された手法を考えるとおもしろいと思います。

● 得点計算のアルゴリズム

役のチェックができたら次は得点を計算します。麻雀では，できた役に従って値が付けられている「ハン(ファン)」があります。またそれとは別に，手牌やあがったときの状況から得る「符」というものもあります。この「符」と「ハン」を得ることで，点数を計算します。

◆ハンと符の値を得る

「ハン」のほうは役を引数としたテーブルにすれば簡単に取得することができます。役によってはメンゼンかどうかでハン数が変わるので，そのあたりも考慮しておきます。それらを考慮して書かれたのがList 2G-3，2G-4(P193)です。

ここで問題となるのは符の数え方です。符の対象となるのは「コーツ」と「カンツ」で，しかも牌の種類によって値が変わります。この計算は役の判定のときにいっしょに行うとムダなコードが省けます。次に待ちやあがりの形を調べて符に加えます。符の値そのものを得るときは牌の状態を引数にしたテーブルから取り出すようにします。これにすべてのあがりに加えられる20符を加えて，さらに符に1ケタの数が付いたならそれを切り上げるなどといった処理を行えば，総合的な符がわかります。

◆総合得点を計算する

得た符とハンから総合得点を得るには，計算するのではなく2次元配列にしたテーブルを使います。このほうが満貫以降の点数も含めて点を取り出すことができて便利です(Fig. 2G-6)。親のときの得点は子の1.5倍になるのですが，単純な1.5倍ではなく半端な数は切り捨てたりしなければなりません。そこで，親のときは子から計算するのではなく，別に作った親用のテーブルから値を引くようにしておきます。

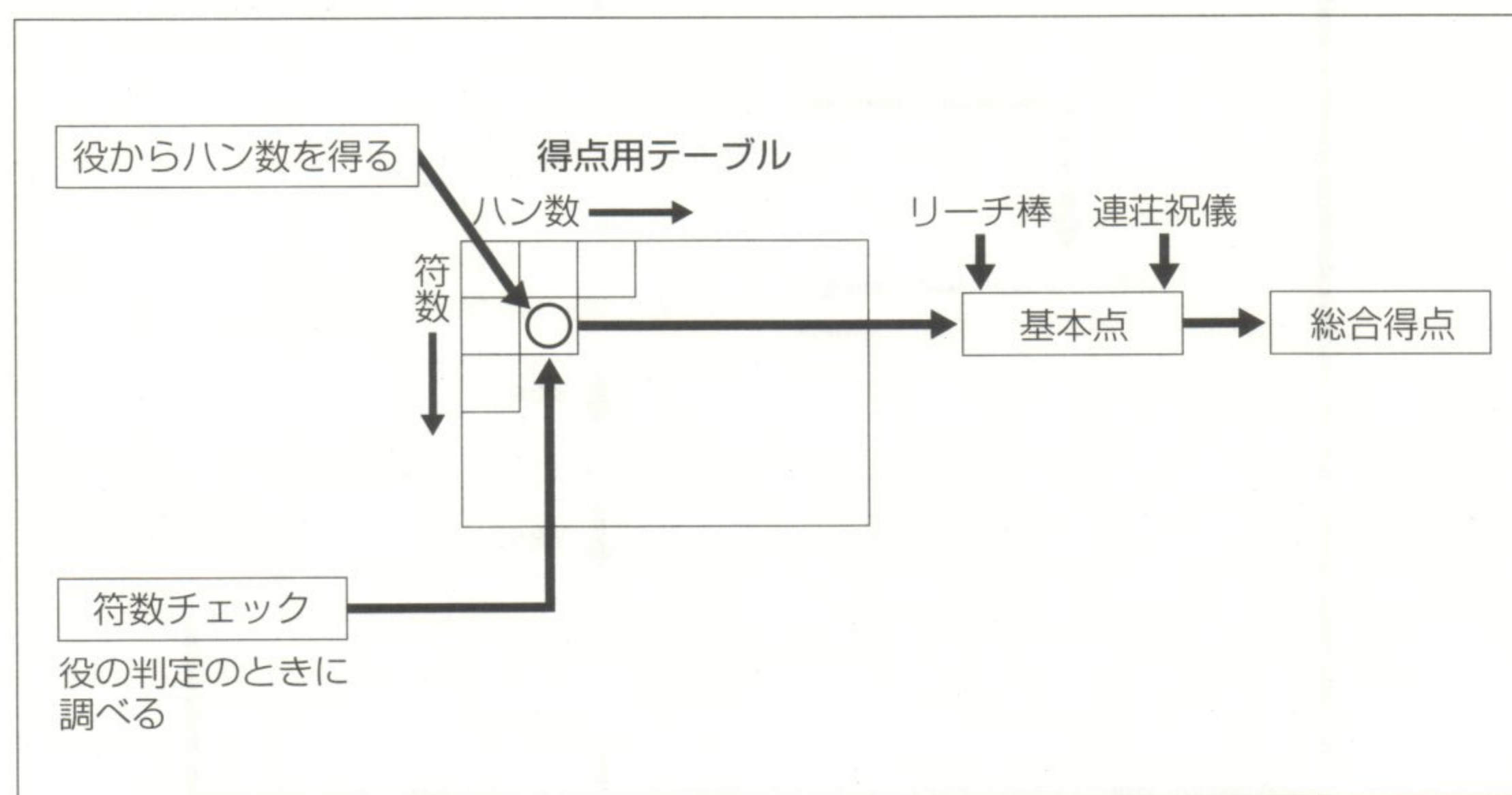



Fig. 2G-6●2次元配列にした得点用のテーブルを用意し，そこからハンと符を基にして基本点を取り出す。その値に付加すべき得点を加えて，総合得点とする

また場にリーチ棒が出ていたときはそれも得点となります。親が連荘しているときはその連荘の数に従って300点単位で値が加わります。この処理も必要です。

こうしてやっと最終的に得た点数がわかります。なかなかめんどうですが，テーブルを活用することでコードそのものは比較的楽になります。

● 麻雀ゲームの思考ルーチン

思考ルーチンは「作りやすい役の判断」と「牌を切る」という2つの作業から構成されています。基本的には普通に麻雀をしている人なら誰でも考えていることを，そのままアルゴリズムとして実現させるだけです(Fig. 2G-7)。

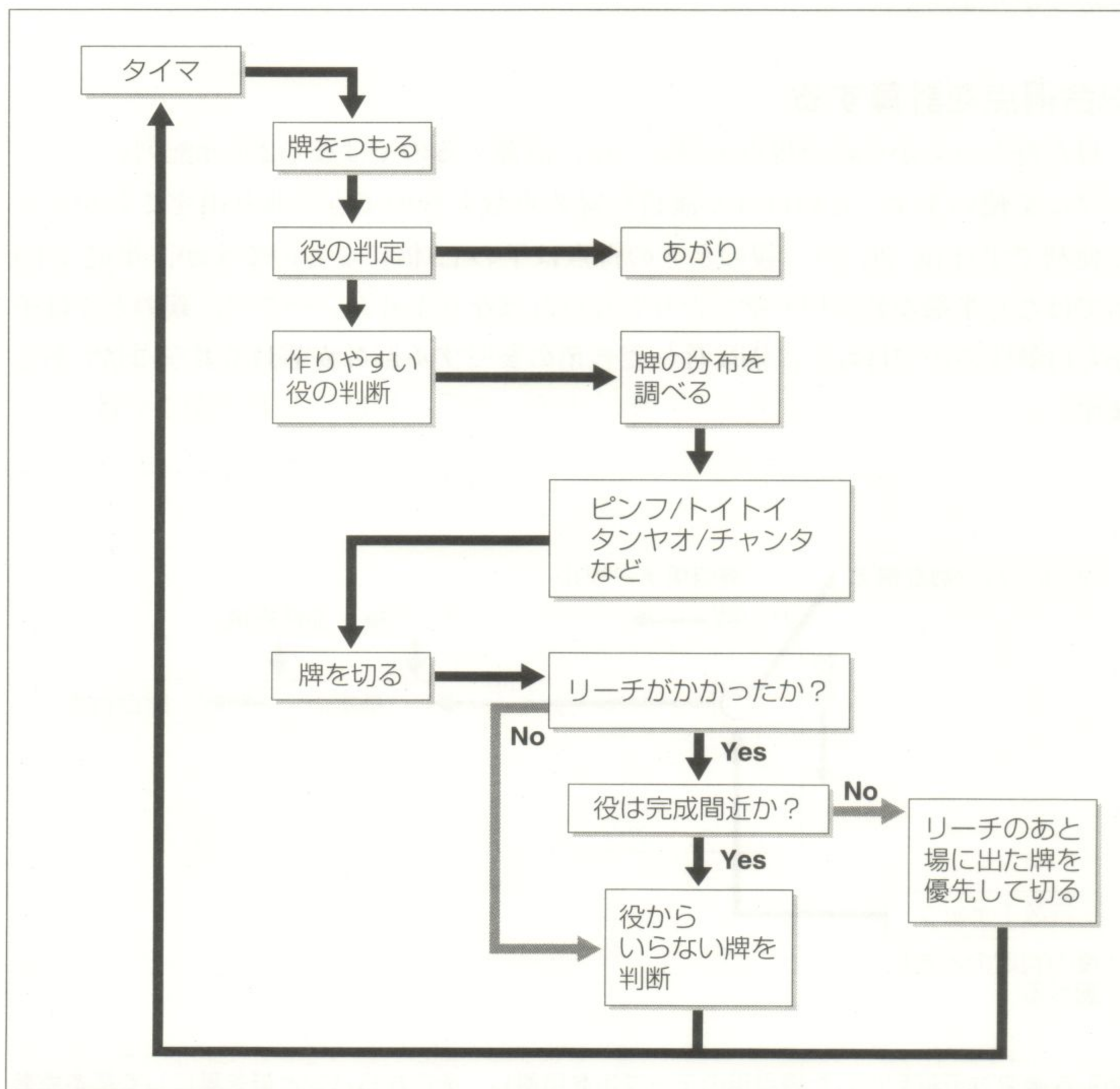



Fig. 2G-7 ●麻雀ゲームの思考ルーチン

まず最初の段階では，「もっとも作りやすい役」を牌の分布状況から調べます。もし1種類の牌が手牌の70%以上を占めるのであれば，その種類の牌でチンイツ系の役にしてみます。このあたりを微妙に変えることでプレイヤーの性格が出てきます。

◆牌の切り方

目指す役が決まったら，それに合わせて牌を切ります。先ほどの例ならチンイツにしたい種類以外の牌を切っていくことになります。あとは切り方を選択していくだけです。「浮いている牌」「見込みのなさそうな待ち方」「余っている字牌」など，切る牌の順序を体系化してコードにします。また，ほかのプレイヤーからリーチがかかったりしたときは，あがるのをあきらめることも判断しなくてはいけません。もしあきらめるのなら，相手が場に出した牌を優先して捨てていくことになります。

◆強いアルゴリズムを作る方法

意外と何でもないように思えますが，これが基本です。あとは「いかにして効率よく計算するか」を詰めていきます。牌の分布を調べるときに，その時点で「字牌」や「浮いている牌」などがわかるので，そうした牌があった時点で調査を打ち切ります。こうすると計算時間とデータを参照する回数が省けます。また役は明確なものではなく「タンヤオ系」か「チャンタ系」程度のおおまかな分け方で調べます。

さらに強い思考ルーチンがほしいときは，「学習させる」「コンピュータで自動的にアルゴリズムを作らせる」ことをします。学習させるときは，ゲームに負けたときに「負けた原因」を考え，それを次の局に反映させるようにします。さらにこうしたアルゴリズムをコンピュータ同士で闘わせて，「負けたらアルゴリズムを少し変える」ことを繰り返して「自然淘汰」的なことをさせると，勝手に強いルーチンができあがります。実現させるには，コードに相当する「条件」部分をデータとして記述しなければなりません。おもしろく楽な方法ですが，つき詰めていくと性格的なものが失われてしまいがちになるので，多少偏りを持たせたほうがよいでしょう。

サンプルゲームの遊び方

本格的な4人制のゲームにしてみました。ルールは「ありあり」で普段私と知り合いたちがやっている「沙姫さんちルール」というものです。あまり基本的なルールからは外れていないと思います。

思考ルーチンは，ここで解説したものをそのまま使っています。個性を出すために「ピンフを優先させる」「タンヤオが好き」といった程度の差を付けています。配牌時の牌の偏りは手を加えていません。役の判定といった具体的なルーチンはこのサンプルゲームを参考にしてください。

*

その昔，私は一局で多牌小牌を繰り返すというチョンボの巣窟でした。知り合いの薦めでコンピュータ麻雀ゲームで何度も繰り返し遊ぶことでそこそこうまくなりました。いまは「強い奴に会いにいく」という状態なので，もしお手合わせしてもいいという方がいらっしゃるようでしたらご連絡ください(笑)。

List 2G-1 ●牌のデータ構造（Delphi）

```
{ 定数宣言 }
const
    PAITBLMAX        = 136;
    POSITIONYAMA     = 0;   { 牌は山にある                  }
    POSITIONTE       = 1;   { 牌は手にある                  }
    POSITIONNAKI     = 2;   { 牌はなかれて場にさらされている }
    POSITIONBA       = 3;   { 牌は場に捨てられた            }
    POSITIONDORA     = 4;   { 牌はドラ表示牌                }

{ 型宣言 }
type
    TPaiData = record              { TPaiData 構造体の定義 }
        Category: integer;         { 種類                  }
        Attribute: integer;        { 属性(字牌など)        }
        Number: integer;           { 数                    }
        Position: integer;         { 置かれている場所      }
    end;
    TPaiDataTbl = array[0..PAITBLMAX] of TPaiData;
    pTPaiDataTbl = ^TPaiDataTbl;


{ 変数宣言 }
var
    { 牌データのテーブル }
    PaiDataTbl: array[0..PAITBLMAX] of TPaiData;
```

List 2G-2●牌のデータ構造(C/C++)

```
/* 定数宣言 */
#define PAITBLMAX          136
#define POSITIONYAMA       0    /* 牌は山にある                    */
#define POSITIONTE         1    /* 牌は手にある                    */
#define POSITIONNAKI       2    /* 牌はなかれて場にさらされている  */
#define POSITIONBA         3    /* 牌は場に捨てられた              */
#define POSITIONDORA       4    /* 牌はドラ表示牌                  */


/* 型宣言 */
typedef struct {            /* TPaiData 構造体の定義    */
    int Category;           /* 種類                      */
    int Attribute;          /* 属性(字牌など)           */
    int Number;             /* 数                        */
    int Position;           /* 置かれている場所          */
} TPaiData;

typedef TPaiData TPaiDataTbl[PAITBLMAX + 1];
typedef TPaiDataTbl *pTPaiDataTbl;


/* 変数宣言 */
/* 牌データのテーブル */
TPaiData PaiDataTbl[PAITBLMAX + 1];
```

List 2G-3●得点計算(Delphi)

```
{ 定数宣言 }
const
  { 子のツモ }
  ScoreTbl_kotsumo: array[0..12,0..7] of LongInt = (
    (    0,   300,   400,   400,   500,   600,   700,   800),
    (  400,   500,   700,   800,  1000,  1200,  1300,  1500),
    (  700,  1000,  1300,  1600,  2000,  2000,  2000,  2000),
    ( 1300,  2000,  2000,  2000,  2000,  2000,  2000,  2000),
    ( 2000,  2000,  2000,  2000,  2000,  2000,  2000,  2000),
    ( 3000,  3000,  3000,  3000,  3000,  3000,  3000,  3000),
    ( 3000,  3000,  3000,  3000,  3000,  3000,  3000,  3000),
    ( 4000,  4000,  4000,  4000,  4000,  4000,  4000,  4000),
    ( 4000,  4000,  4000,  4000,  4000,  4000,  4000,  4000),
    ( 4000,  4000,  4000,  4000,  4000,  4000,  4000,  4000),
    ( 6000,  6000,  6000,  6000,  6000,  6000,  6000,  6000),
    ( 6000,  6000,  6000,  6000,  6000,  6000,  6000,  6000),
    ( 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000)
  );
```

▼

List 2G-3

```
{ 親のツモ }
ScoreTbl_oyatsumo: array[0..12,0..7] of LongInt = (
  (    0,   500,   700,   800,  1000,  1200,  1300,  1500),
  (  700,  1000,  1300,  1600,  2000,  2300,  2600,  2900),
  ( 1300,  2000,  2600,  3200,  3900,  4000,  4000,  4000),
  ( 2600,  3900,  4000,  4000,  4000,  4000,  4000,  4000),
  ( 4000,  4000,  4000,  4000,  4000,  4000,  4000,  4000),
  ( 6000,  6000,  6000,  6000,  6000,  6000,  6000,  6000),
  ( 6000,  6000,  6000,  6000,  6000,  6000,  6000,  6000),
  ( 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000),
  ( 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000),
  ( 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000),
  (12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000),
  (12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000),
  (16000, 16000, 16000, 16000, 16000, 16000, 16000, 16000)
);

{ 子のロンあがり }
ScoreTbl_koron: array[0..12,0..7] of LongInt = (
  ( 1000,  1300,  1600,  2000,  2300,  2600,  2900,  3200),
  ( 2000,  2600,  3200,  3900,  4500,  5200,  5800,  6400),
  ( 3900,  5200,  6400,  7700,  8000,  8000,  8000,  8000),
  ( 7700,  8000,  8000,  8000,  8000,  8000,  8000,  8000),
  ( 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000),
  (12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000),
  (12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000),
  (16000, 16000, 16000, 16000, 16000, 16000, 16000, 16000),
  (16000, 16000, 16000, 16000, 16000, 16000, 16000, 16000),
  (16000, 16000, 16000, 16000, 16000, 16000, 16000, 16000),
  (24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000),
  (24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000),
  (32000, 32000, 32000, 32000, 32000, 32000, 32000, 32000)
);

{ 親のロンあがり }
ScoreTbl_Oyaron: array[0..12,0..7] of LongInt = (
  ( 1500,  2000,  2400,  2900,  3400,  3900,  4400,  4800),
  ( 2900,  3900,  4800,  5800,  6800,  7700,  8700,  9600),
  ( 5800,  7700,  9600, 11600, 12000, 12000, 12000, 12000),
  (11600, 12000, 12000, 12000, 12000, 12000, 12000, 12000),
  (12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000),
  (18000, 18000, 18000, 18000, 18000, 18000, 18000, 18000),
  (18000, 18000, 18000, 18000, 18000, 18000, 18000, 18000),
  (24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000),
  (24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000),
  (24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000),
  (36000, 36000, 36000, 36000, 36000, 36000, 36000, 36000),
  (36000, 36000, 36000, 36000, 36000, 36000, 36000, 36000),
  (42000, 42000, 42000, 42000, 42000, 42000, 42000, 42000)
);
```

```
  ScoreAttrOyatsumo      = 0;
  ScoreAttrOyaron        = 1;
  ScoreAttrKotsumo_ko    = 2;
  ScoreAttrKotsumo_oya   = 3;
  ScoreAttrKoron         = 4;

{ 得点取得 }
{ Han          = ハン数 }
{ Fu           = 符数 }
{ ScoreAttr    = あがり方 }
{ RenchanCount = 連荘数 }
{ FlagChiitoi  = チートイツかどうか }
function GetScore(Han, Fu, ScoreAttr, RenchanCount: integer;
  FlagChiitoi: boolean): LongInt;
var
  TmpScore: LongInt;
  HanCount: integer;
begin
    Fu := Fu div 10;
    { 符数の調整 }
    if (ScoreAttr = ScoreAttrKoron) or
                           (ScoreAttr = ScoreAttrOyaron) then
        Fu := Fu - 3
    else
        Fu := Fu - 2;
    { チートイツかどうか }
    if FlagChiitoi then begin
        if Han <= 4 then begin
            if Han <= 3 then
                Dec(Han);
            { チートイツなら計算を別に }
            case ScoreAttr of
              ScoreAttrOyatsumo:     TmpScore := 1600 * Han;
              ScoreAttrOyaron:       TmpScore := 2400 * Han;
              ScoreAttrKotsumo_ko:   TmpScore := 800 *  Han;
              ScoreAttrKotsumo_oya:  TmpScore := 1600 * Han;
              ScoreAttrKoron:        TmpScore := 1600 * Han;
            end;
            result := TmpScore;
            exit;
        end else begin
            Fu := 7;
        end;
    end;
    { ハン数の調整 }
    { 満貫以上は別計算として切り詰める }
    Dec(Han);
    if Han >= 13 then
        Han := 12;
```

List 2G-3

```
    { 得点取得 }
    case ScoreAttr of
      ScoreAttrOyatsumo:      TmpScore := ScoreTbl_oyatsumo[Han,Fu];
      ScoreAttrOyaron:        TmpScore := ScoreTbl_oyaron[Han,Fu];
      ScoreAttrKotsumo_ko:    TmpScore := ScoreTbl_kotsumo[Han,Fu];
      ScoreAttrKotsumo_oya:   TmpScore := ScoreTbl_oyatsumo[Han,Fu];
      ScoreAttrKoron:         TmpScore := ScoreTbl_koron[Han,Fu];
    end;
    { 連荘していた場合の得点追加 }
    if RenchanCount <> 0 then begin
        if (ScoreAttr = ScoreAttrOyaron) or
                                  (ScoreAttr = ScoreAttrKoron) then begin
            TmpScore := TmpScore + (RenchanCount * 300);
        end else begin
            TmpScore := TmpScore + (RenchanCount * 100);
        end;
    end;
    result := TmpScore;
end;
```

List 2G-4 ●得点計算（C/C++）

```
/* 子のツモ */
long ScoreTbl_kotsumo[12+1][7+1] = {
  {    0,   300,   400,   400,   500,   600,   700,   800},
  {  400,   500,   700,   800,  1000,  1200,  1300,  1500},
  {  700,  1000,  1300,  1600,  2000,  2000,  2000,  2000},
  { 1300,  2000,  2000,  2000,  2000,  2000,  2000,  2000},
  { 2000,  2000,  2000,  2000,  2000,  2000,  2000,  2000},
  { 3000,  3000,  3000,  3000,  3000,  3000,  3000,  3000},
  { 3000,  3000,  3000,  3000,  3000,  3000,  3000,  3000},
  { 4000,  4000,  4000,  4000,  4000,  4000,  4000,  4000},
  { 4000,  4000,  4000,  4000,  4000,  4000,  4000,  4000},
  { 4000,  4000,  4000,  4000,  4000,  4000,  4000,  4000},
  { 6000,  6000,  6000,  6000,  6000,  6000,  6000,  6000},
  { 6000,  6000,  6000,  6000,  6000,  6000,  6000,  6000},
  { 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000},
};

/* 親のツモ */
long ScoreTbl_oyatsumo[12+1][7+1] = {
  {    0,   500,   700,   800,  1000,  1200,  1300,  1500},
  {  700,  1000,  1300,  1600,  2000,  2300,  2600,  2900},
  { 1300,  2000,  2600,  3200,  3900,  4000,  4000,  4000},
  { 2600,  3900,  4000,  4000,  4000,  4000,  4000,  4000},
  { 4000,  4000,  4000,  4000,  4000,  4000,  4000,  4000},
  { 6000,  6000,  6000,  6000,  6000,  6000,  6000,  6000},
  { 6000,  6000,  6000,  6000,  6000,  6000,  6000,  6000},
```

```
  { 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000},
  { 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000},
  { 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000},
  {12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000},
  {12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000},
  {16000, 16000, 16000, 16000, 16000, 16000, 16000, 16000},
};

/* 子のロンあがり */
long ScoreTbl_koron[12+1][7+1] = {
  { 1000,  1300,  1600,  2000,  2300,  2600,  2900,  3200},
  { 2000,  2600,  3200,  3900,  4500,  5200,  5800,  6400},
  { 3900,  5200,  6400,  7700,  8000,  8000,  8000,  8000},
  { 7700,  8000,  8000,  8000,  8000,  8000,  8000,  8000},
  { 8000,  8000,  8000,  8000,  8000,  8000,  8000,  8000},
  {12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000},
  {12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000},
  {16000, 16000, 16000, 16000, 16000, 16000, 16000, 16000},
  {16000, 16000, 16000, 16000, 16000, 16000, 16000, 16000},
  {16000, 16000, 16000, 16000, 16000, 16000, 16000, 16000},
  {24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000},
  {24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000},
  {32000, 32000, 32000, 32000, 32000, 32000, 32000, 32000},
};

/* 親のロンあがり */
long ScoreTbl_Oyaron[12+1][7+1] = {
  { 1500,  2000,  2400,  2900,  3400,  3900,  4400,  4800},
  { 2900,  3900,  4800,  5800,  6800,  7700,  8700,  9600},
  { 5800,  7700,  9600, 11600, 12000, 12000, 12000, 12000},
  {11600, 12000, 12000, 12000, 12000, 12000, 12000, 12000},
  {12000, 12000, 12000, 12000, 12000, 12000, 12000, 12000},
  {18000, 18000, 18000, 18000, 18000, 18000, 18000, 18000},
  {18000, 18000, 18000, 18000, 18000, 18000, 18000, 18000},
  {24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000},
  {24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000},
  {24000, 24000, 24000, 24000, 24000, 24000, 24000, 24000},
  {36000, 36000, 36000, 36000, 36000, 36000, 36000, 36000},
  {36000, 36000, 36000, 36000, 36000, 36000, 36000, 36000},
  {42000, 42000, 42000, 42000, 42000, 42000, 42000, 42000},
};

#define ScoreAttrOyatsumo      0
#define ScoreAttrOyaron        1
#define ScoreAttrKotsumo_ko    2
#define ScoreAttrKotsumo_oya   3
#define ScoreAttrKoron         4

/* 得点取得 */
/* Han          = ハン数 */
```

List 2G-4

```
/* Fu           = 符数 */
/* ScoreAttr    = あがり方 */
/* RenchanCount = 連荘数 */
/* FlagChiitoi  = チートイツかどうか */
long GetScore(int Han, int Fu, int ScoreAttr, int RenchanCount,
  bool FlagChiitoi)
{
    long TmpScore;
    int HanCount;

    Fu /= 10;
    // 符数の調整
    if ((ScoreAttr == ScoreAttrKoron) ||
                              (ScoreAttr == ScoreAttrOyaron)) {
        Fu -= 3;
    } else {
        Fu -= 2;
    }
    // チートイツかどうか
    if (FlagChiitoi) {
        if (Han <= 4) {
            if (Han <= 3)
                Han--;
            // チートイツなら計算を別に
            switch (ScoreAttr) {
              case ScoreAttrOyatsumo:
                TmpScore := 1600 * Han;
                break;
              case ScoreAttrOyaron:
                TmpScore := 2400 * Han;
                break;
              case ScoreAttrKotsumo_ko:
                TmpScore := 800 *  Han;
                break;
              case ScoreAttrKotsumo_oya:
                TmpScore := 1600 * Han;
                break;
              case ScoreAttrKoron:
                TmpScore := 1600 * Han;
                break;
            }
            return TmpScore;
        } else {
            Fu = 7;
        }
    }
    // ハン数の調整
    // 満貫以上は別計算として切り詰める
    Han--;
    if (Han >= 13)
```

```
        Han = 12;
    // 得点取得
    switch (ScoreAttr) {
      case ScoreAttrOyatsumo:
        TmpScore = ScoreTbl_oyatsumo[Han][Fu];
        break;
      case ScoreAttrOyaron:
        TmpScore = ScoreTbl_oyaron[Han][Fu];
        break;
      case ScoreAttrKotsumo_ko:
        TmpScore = ScoreTbl_kotsumo[Han][Fu];
        break;
      case ScoreAttrKotsumo_oya:
        TmpScore = ScoreTbl_oyatsumo[Han][Fu];
        break;
      case ScoreAttrKoron:
        TmpScore = ScoreTbl_koron[Han][Fu];
        break;
    }
    // 連荘していた場合の得点追加
    if (RenchanCount != 0) {
        if ((ScoreAttr == ScoreAttrOyaron) ||
          (ScoreAttr = ScoreAttrKoron) {
          TmpScore += RenchanCount * 300;
        } else {
          TmpScore += RenchanCount * 100;
        }
    }
    return TmpScore;
}
```

Section 8 格闘ゲーム

キャラクタ同士が拳と拳を交え合う格闘ゲームは，アクション系ゲームに関するほとんどの手法が組み合わせて作られています。その作り方を述べてみたいと思います。

● 闘って敵を倒す

いま実は非常に危険な状態です。ことの発端は，あるアニメ制作関係者2人が私を見て「そういえば声優の○山さんに似ている」と話したことだったと思います。この女性の声優さんは，あるアニメのイベントでみずから声を当てたキャラクタの格好に扮していました。そしてあるきっかけのせいで，こともあろうに私にその格好をしろと知人たちがいってきたのです。これがまたミニスカでパンチラありのとても恥ずかしい格好です。「写真を撮って売れば服代ぐらいはペイできる」と数々の陰謀が張りめぐらされていて，これ以上の身の危険はありません(笑)。

そこで「身の危険」というものを体感することができる「格闘ゲーム」について取りあげてみます。なお3次元ポリゴンではなく2次元での利用を中心にしましたが，画面表示に関する処理を除き，考え方は3次元でも同じになります。

格闘ゲームは「1つのゲーム」としてすでにジャンルが成り立っています。「敵を倒す」というもっとも単純で刺激的な内容のせいか，「味付け」ともいえるバリエーションがやや乏しくなりつつあります。今後は格闘ゲーム単体よりも，「ミニゲーム」と呼ばれるものとなって，ロールプレイングゲームやシミュレーションゲームなどに吸収されてしまうことも考えられます。最近の複雑化するゲームの反動が起きることもあり得そうです。いずれにしろ，近いうちに大きな変化が起きるかもしれません。

● 格闘ゲームとは？

「2人のキャラクタが登場して闘う」というのが格闘ゲームの大まかな定義となります。闘うときは特定のキーにより「パンチ」「キック」などが出ますが，さらにキーの組み合わせで特殊な「必殺技」などを出すことができます。

格闘ゲームは，もともとはアクションゲームと呼ばれる分野から派生したゲームです。アクションゲームはアクションロールプレイングゲームなどとも呼ばれていました。これらゲームは画面表示の視点はキャラクタや背景を横から見た状態で，キャラクタが画面の端にさしかかると表示がスクロールしていきます。この点はロールプレイングゲームの画面処理と似たような感じですが，多くの敵キャラクタが次から次へと出てくるのを倒していくことでゲームを進めるのは，シューティングゲームと似た仕組みです。初期に作られた格闘ゲームの始祖となるものは，多くの場合，この要素を持っていました。このなかから「決まった相手と闘うこと」だけを抜き出し，ボクシングや柔道の試合などに見られる「時間制限」「ラウンド制」などを加えたのが現在の格闘ゲームということになります。

◆格闘ゲームの魅力

このゲームのおもしろさのポイントは，その「暴力性」だけに見られがちです。たしかに敵キャラクタを倒すことによる爽快感はありますが，実際にはそれだけではありません。すでに成熟した分野であるシューティングゲームと対比させてみると，それがよりはっきりとわかります。たとえば格闘ゲームでの1対1の闘いは，シューティングゲームのボスキャラとの闘いに相当します。シューティングゲームでの「クライマックス」とも呼べる部分がすぐに楽しめるのも格闘ゲームの特徴です。

シューティングゲームでは，キャラクタそのものに人気が出ることはあまりありません。一方格闘ゲームでは，各キャラクタが絶大な人気を持っています。人と人が闘うところから感じられるヒーロー性ともいえますが，人というキャラクタだからこそできる「感情移入」といった面もかなり大きく存在します。アクションゲームよりも強い攻撃性を持たせたことで，これがさらに高まる結果となっています。

感情移入を膨らませる要素はほかにも多くあります。その1つとしてストーリ性を持たせているゲームもあります。内容としては「各キャラクタはそれぞれ目的を持っていて，闘いを重ねていくことでそれが達成される」といったものが多くを占めているようです。また技の出し方も複雑化しています。技と技を組み合わせるとより強力な攻撃になったり，体力がないときにしか出せない必殺技などです。難しい組み合わせを使いこなすという一種のパズル的楽しさともいえるでしょう。

◆さまざまな要素が複雑に絡み合う

こうした単純な「敵を倒す」という面に加えて，ほかの楽しさが複雑に組み合わされているのが，格闘ゲームのポイントとなっています。かつて人気のあったゲ

ームはこうした点をうまく使っています。現在作られている格闘ゲームでは「ストーリ」や「キャラクタ」「画面の華やかさ」「技の変化や難しさ」といった面で差を出そうとしているところがほとんどになっています。

これらの楽しさを支えているのが「リアルな画面処理」です。キャラクタの動きがぎこちなければ，こうした感情移入は生まれにくいものです。それではその方法を調べてみることにしましょう。

●格闘ゲームの中身

実際に画面へ表示される内容から，プログラムの動きを探ってみることにします。

ゲーム中の画面はだいたい(Fig. 2H-1)のようなものです。表示そのものは「キャラクタ」を表現するグラフィックの「パーツ」と「背景」に分かれています。実際の画面はパーツと背景を「合成」したものとなっています。プログラムでの処理もそれぞれ別に扱います。

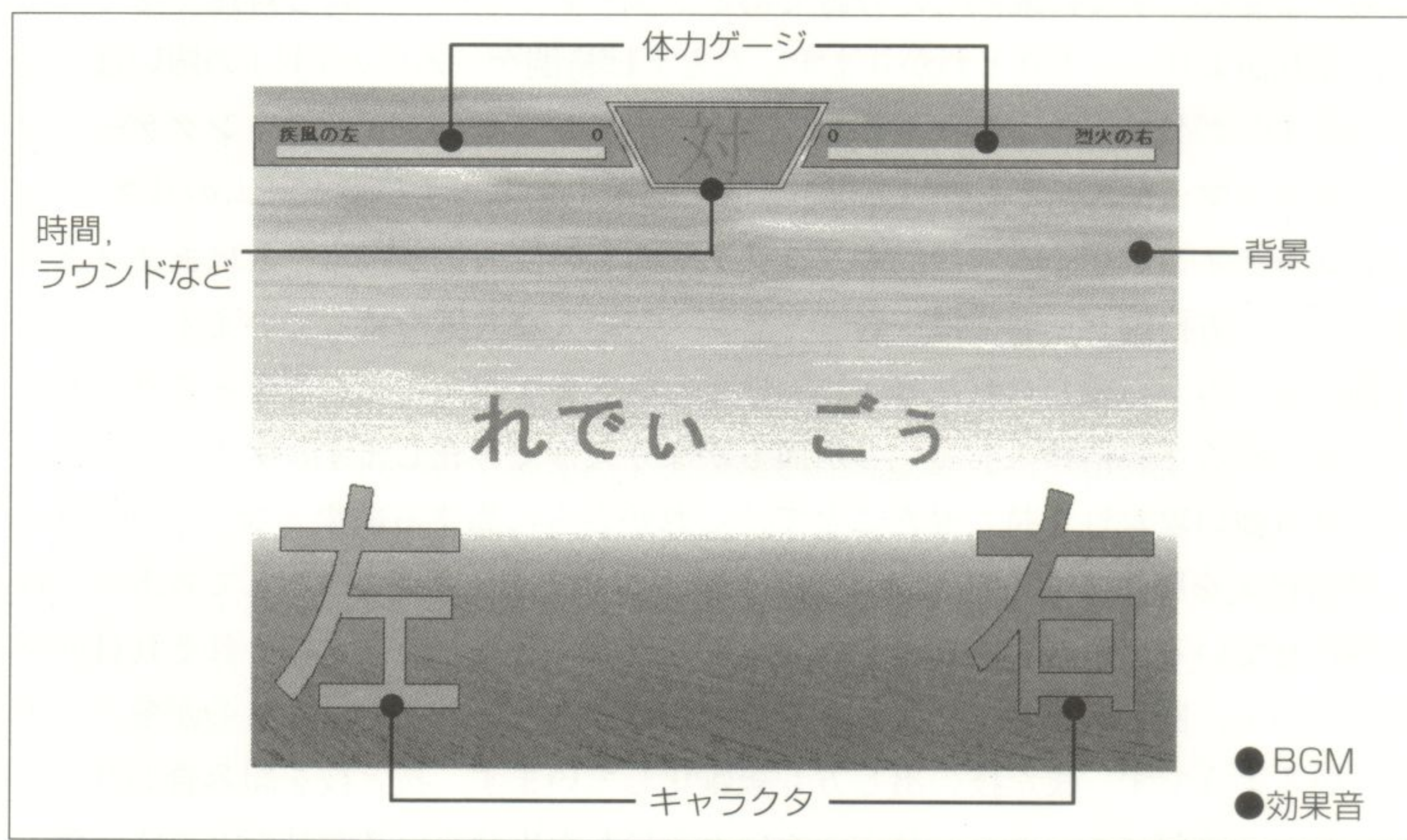



Fig. 2H-1 ●格闘ゲームの画面構成。ゲームプレイ時にはBGMと効果音が加わる

◆キャラクタの移動

移動用のキーを押すとキャラクタが指定された方向へと動きます。左右方向ならキャラクタが歩くようなグラフィックが表示されます。これをプログラム上のデ

ータにするには，パーツの「アニメーションデータ」とそれぞれのキーが対応した形になっているはずです。さらにアニメーションの動きなどを指定するために「パーツをどの順番で表示するか」を決める「スクリプトデータ」もあります。

◆当たり判定

敵に殴られたり必殺技が当たると，体力ゲージがその分だけ下がります。殴られたかどうかを調べるために「当たり判定(衝突判定)」を行っています(Fig. 2H-2)。スプライトで色の衝突がわかる場合には，普通はそれを利用します。こうした機能がないなら，キャラクタが移動する最小の量である「移動量」ごとに区切ったマス目へキャラクタを投影してそこで判断したり，「キャラクタのいる座標と大きさを基にして重なるかどうかを判断する」「数学的に物体が重なっているか判断する」といった方法もあります。

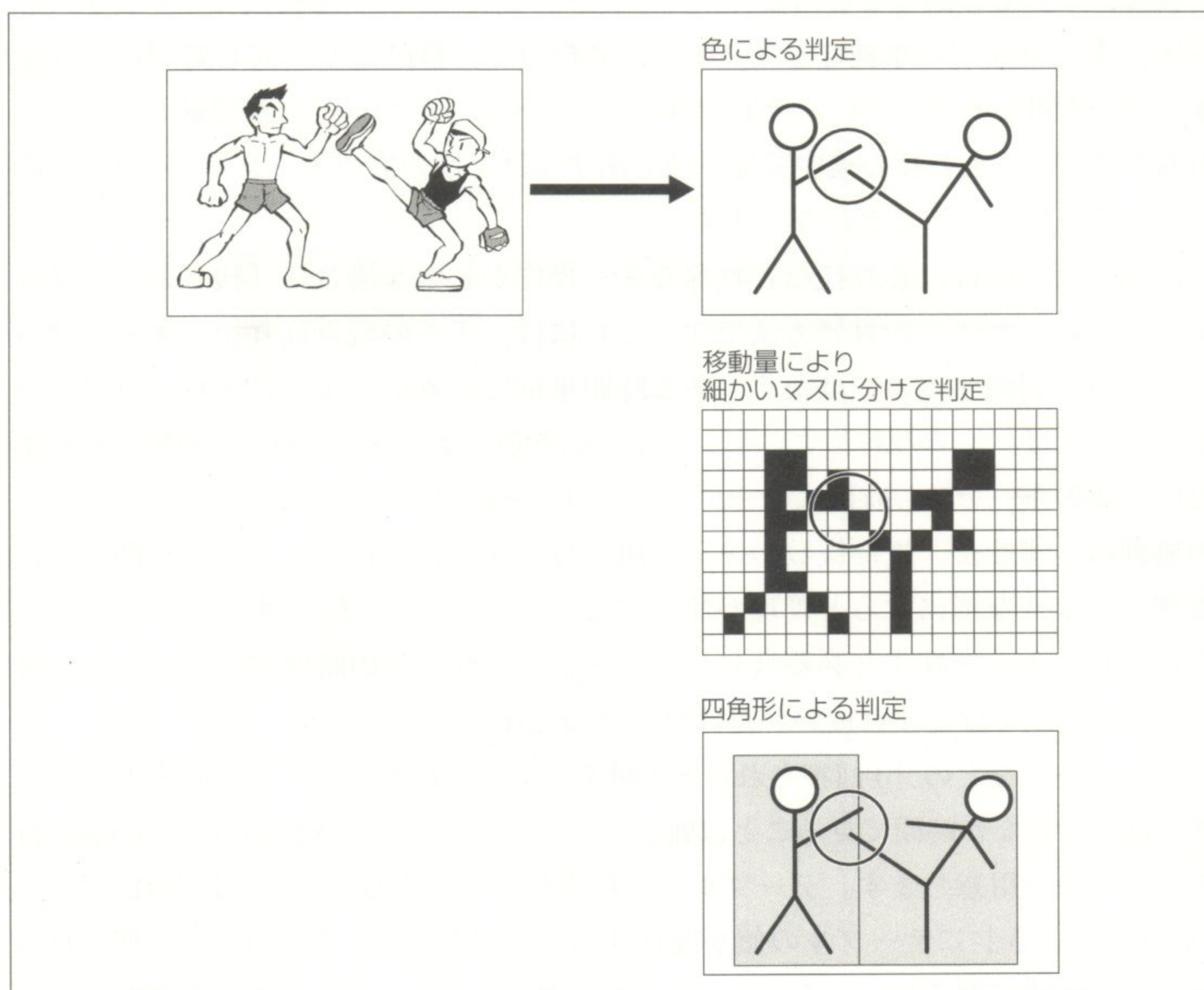



Fig. 2H-2●3種類の当たり判定。色の重なりによって判定する方法，移動量ごとに区切ったマス目にキャラクタを投影する方法，キャラクタの周囲を四角形に区切その四角形同士の接触で判定する方法

◆ダメージの判定

「どれだけダメージを与えたか」といった値は，攻撃の種類と対になったテーブルを使って得ます。闘うキャラクタが変わっても参照するテーブルを変えるだけで，それぞれの技に対するダメージの値をコード部分の変更なく簡単に得ることができます。技とキーコードの対応部分も同様にテーブルにして管理します。あとはダメージを受けたほうの体力ゲージが0になったのならゲーム終了です。

◆技の処理

ここまでは，アクションゲームやシューティングゲームの分野で使われている手法がそのまま利用されています。それではもっとも格闘ゲームらしい処理はどこかというと，「技の処理」と「コンピュータ側キャラクタの思考ルーチン」といったところにあります。

技は，これも移動などと同じ「スクリプト」としてデータを持たせています。パンチ，キックなどは単純な1つのキーで済むので，移動などと同じ処理になります。キーが押されるたびに，それぞれのアニメーションパターンが連続して表示されるだけです。もし技の途中で違う技を出すには「技のアニメーションパターンが全部出るまで待つ」ことになります。

問題となるのは，必殺技など特殊なキー操作が必要な場合に「複数のキーが同時に押されたとき」の対処方法です。これには，キーの読み取りを，キーが押された時点で判断するのではなく，ある時間単位で読み取るようにすれば解決します。たとえばパンチのあとすぐキックという状態では，パンチのキーが押されたあとに，次のキーを読む時間になってキックのキーが押されていないようならキックの処理はしません。また複数のキーを組み合わせて使う技でも，この時間ごとに処理するような形にすると管理が簡単になります。「パンチ」「キック」「パンチ」と連続してキーを押すと必殺技になるときは，それぞれの時間で必要なキーが押されているかを確認すればいいだけです(Fig. 2H-3)。

キーボードなどの「1回押されると1回イベントが起きる」タイプの入力デバイスでは，なかなか時間で見ることは難しいものです。このときは使うキーの数だけのテーブルを用意します。テーブルでそれぞれに対応するキーが「押されたとき」「離されたとき」にテーブルの値が変化するようにしてき，1つのキーが押されるたびに，同時に押されているキーがないかを調べます。こうしたキーが連続するときは，「キーが離された時点ですべてのキーが離されている」ことを確認してから次のキーの組み合わせを調べるようにします。この「間」のようなものは，何度

か調整することになります。

◆思考ルーチン

シューティングゲームでは，敵となるキャラクタの動きは，出現する「位置」と「時間」それに「決まった動き方」などをあらかじめ設定しておくぐらいでよかったのですが，格闘ゲームではそうもいきません。コンピュータが操作する敵キャラクタのために，ある種の思考ルーチンが必要です。

コンピュータ側キャラクタの動きを決めるときは「相手の移動や動作」に比例

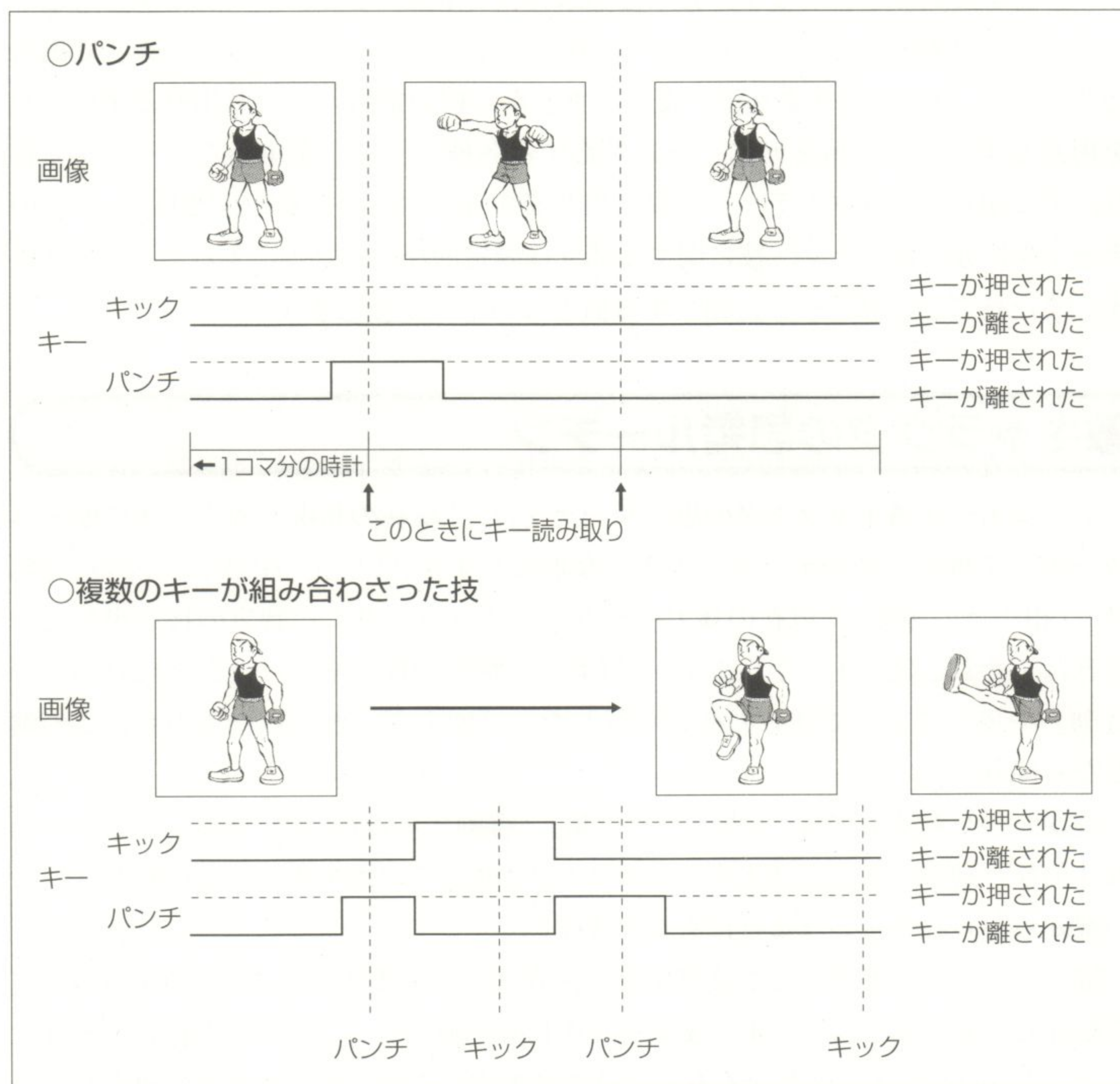



Fig. 2H-3●あらかじめ時間単位を設定しておき，その時間単位ごとに押されているキーを読み取って対応する処理を行う。複数のキーが組み合わさって1つの技が出る場合には，時間単位を複数まとめた一定期間内に押されたボタンを読み取り，その組み合わせに対応する処理を行う

するような形になります。基本的には「何かが」が起きたら「どう対処する」といったことをそれぞれ条件に従って分岐するような形になります。これを「何かの条件がある一定を満たしているのならこちらを優先させる」といった比例条件を簡単に作るため，データとして定義するようにします。

格闘ゲームのデータ構造

画画表示に使うパーツデータは，それぞれ「パーツ用グラフィックデータへのポインタ」を配列の形にします。技などのスクリプトデータは，キーデータと対応させておくと便利です。「パンチ」なら「0」,「パンチ」「キック」「パンチ」で出る必殺技なら「1」というように，まず「キーから技の定数」を取り出せるテーブルを用意します。この値を元に「技に対応する各種データ」を集めたテーブルを作り，そこからスクリプトデータを取り出します(List 2H-1，2H-2(P211))。2つのテーブルに分けることで，技に関する部分は固定的なテーブルにすることができます。また技ごとのダメージに相当する値もこの中で定義します。

敵キャラクタの思考ルーチン

コンピュータ側キャラクタの思考ルーチンは,「現在の状況」から「次に取るべき行動」を得る一種のテーブルのような形にします。「現在の状況」となる「鍵」は,「相手との距離」「現在の体力や時間」です。ランダムに特定の技を出したいときは，それも鍵の1つとします。「相手から特定の攻撃を受けたときに対応する行動」や場合によってはその行動を行わない「ゆらぎ」といったものも，この鍵となる部分です。

鍵から得られるものは,「キックで攻撃」「移動するだけ」「ジャンプ」といった具体的な行動の内容となります。キャラクタの取ることができる行動の数だけ，この鍵から得られるということにもなります。

鍵から行動への変換にはさまざまな方法が考えられますが，ここでは1つの方法を紹介してみることにします。まずそれぞれの行動パターンをデータ化し，それに対して1つのパラメータとなるものを持たせます。このパラメータは「何をしたいか」という「意思」であると考えると明確になるかもしれません。鍵の値によって，それぞれの意思の大きさ(パラメータ)を増減させ，意思がもっとも大きかったものが最終的に「次に取るべき行動」となります。行動自体はスクリプトとし

て定義し，これもテーブルにしておきます(Fig. 2H-4)。

◆しきい値による条件判断

各条件判断処理では，それぞれの意思となるパラメータを操作します。これは単純に「ある1つの条件を基にして，それぞれの意思を変える」というようにします。これらの条件判断には「しきい値」と呼ばれるものが必ず存在します。たとえば「相手のキャラクタが近くにいるかどうか」といったときは，その「近くかそうでないか」を判断するための値が必要です。この値は10だったり5だったりといろいろでしょうが，ある値を「しきい値」として，この値以下なら「近く」そう

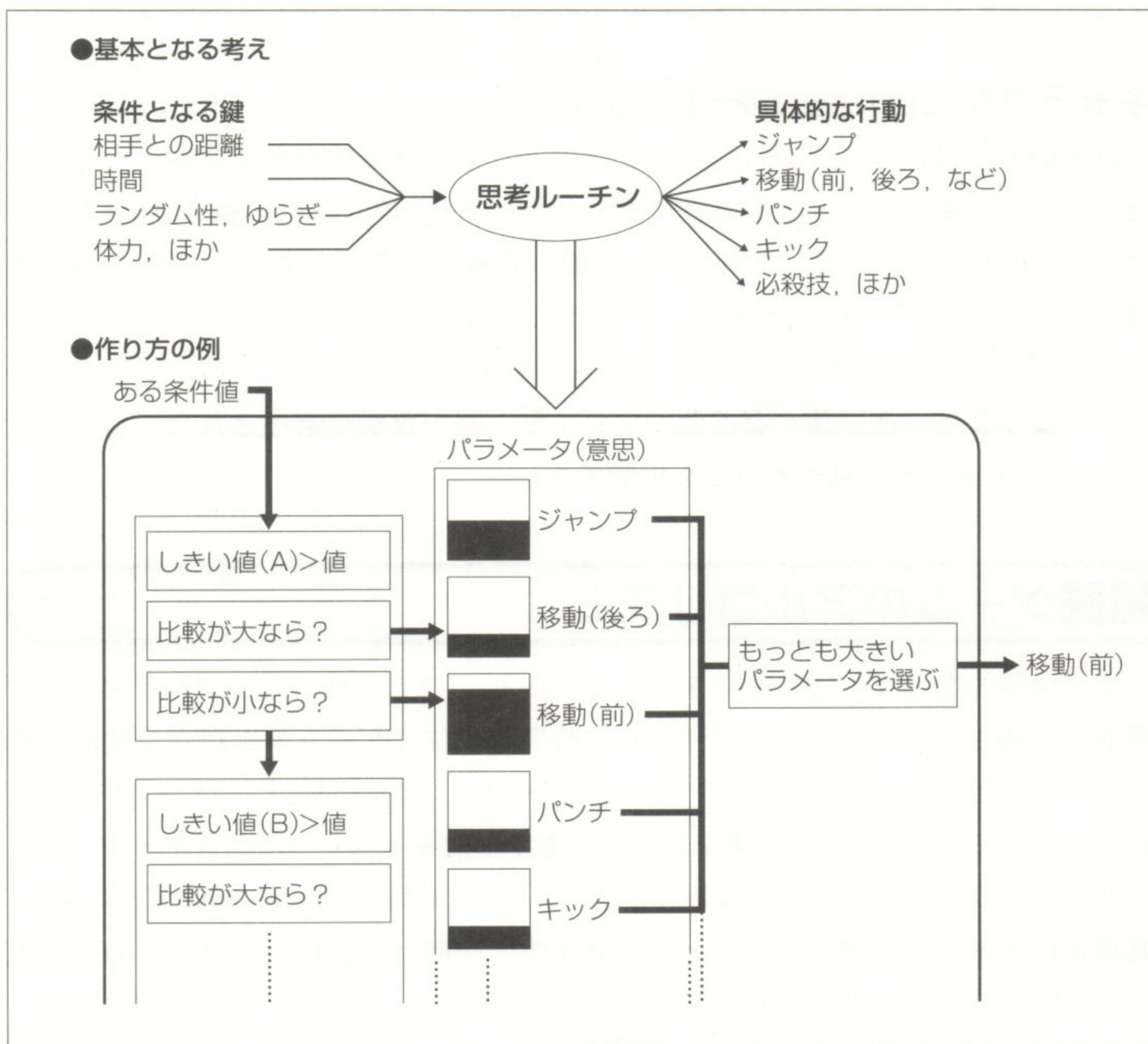



Fig. 2H-4 ●思考ルーチンの処理方法。さまざまな条件を用意し，その条件を元にしてルーチン内で計算して具体的な行動を決める。思考ルーチン内では，条件値を元に行動ごとに設けられたパラメータ値を変化させ，もっともパラメータ値の大きいものを行動として実現する

でなければ「近くではない」と判断できます。さらにその「近く」の度合いより「歩いて近づく」「ジャンプ」などの意思に差を出すことができます。判断したい条件の数だけしきい値を用意しておき，これをゲーム中やキャラクタごとに変動させることにより，キャラクタの個性やより人間らしい動きを出すことができます。

条件判断処理は各条件ごとに関数として作ります。ゲーム場面によっては，必要でない条件も出てくるでしょう。そんなときは必要でない条件の判断処理そのものを外したいこともあります。このため，処理には関数ポインタにして並べたリストの形を取ります。条件を加えたり外したりしたいときは，このリストを操作するようにします。データを処理する部分はこのポインタをつぎつぎとたどっていき，リストが終わった時点で処理を止めます。

◆キャラクタごとに強弱を付ける

以上の方法はけっこう凝った方法ですので，もし速度的な面で問題となる場合は，さらに単純化した方法が使われます。もしキャラクタごとに強さ/弱さといったものを出したいときは，思考ルーチン側の最適化も必要ですが，「判断の時間的遅さ/速さ」といった「反応速度」だけでも差を出すことができます。遅ければ遅いほどプレイヤーの攻撃を受けやすくなり，結果として「弱いキャラクタ」ということになります。逆に速くなると，プレイヤー側が攻撃を避けきれなくなるので「強いキャラクタ」と見せることも可能です。

●格闘ゲームのアルゴリズム

まず全体的な流れを見てください(Fig. 2H-5)。ここで使われる「タイマ」が定期的に起動され，このタイマで使われる時間の単位でゲームの処理が連続的に続いていくことになります。キャラクタの表示でも，この時間ごとにパーツが切り換わることでアニメーション効果を出しています。敵キャラクタの動きや各種スクリプトの読み取り処理もここで行われます。リアルな動きを求める格闘ゲームでは，処理速度が厳しく要求されることになります。今回は，速度を上げるための工夫としてほとんどの処理にテーブルを使用しています。

◆技の判断方法

キーコードなどの入力は，画面更新などといっしょにやってしまうとたいへん短い時間で必要なキーをすべて押さなければならなくなってしまうので，ある程度

「読み取る時間」を持たせなければなりません。実際には「タイマが4回起動したらキーを読み取る」といった内容にして処理を行います。読み取ったキーは，それぞれフラグとして使う変数の形で格納します。このフラグを構造体のように集めて定義します。

技で使われるキーの組み合わせを定義するときには，キー用の構造体がつながったリストになります。技の判断方法は「各段階で共通するもの」を集め，「合うキーがなくなった」段階で「技のスクリプト」を得るツリー形式の構造を取ると検索が楽です。1つのキャラクタが持つ技はそう多くはないのでデータ化は容易です。

検索方法は，最初にキーが押されたら，それと対応するキーをリストの最初に置かれている項目から探します。もし見つかったらその項目へのポインタを保存します。次のキーが押されたときは，項目にあるキーごとに収められたポインタから対応するものを最初と同じように保存します。こうしてキーが押されるたびにポインタをたどっていき，ポインタがなくなったらそのときの項目に定義されている技を実行します(Fig. 2H-6)。技のキー組み合わせが途中で合わなかったり，キーが

Fig. 2H-5 ●格闘ゲームのプログラムの流れ

押されている間隔が空いたようなときは，現在押されているキーに対応する行動を実行して検索に使ったポインタは破棄します。

◆当たり判定の方法

当たり判定の方法として，移動量ごとに判断する方法をシューティングゲームのところで取りあげました。簡単に説明すると，各キャラクタがいる位置に相当する部分を「当たり判定用バッファ」の対応するところへ値として入れます。もし8ドット単位で動くのなら，640×480の画面で，80×60=4800バイト程度のメモリで済みます。これならキャラクタごとにクリアしてもそんなに時間はかかりません。判定するときは当たり判定用バッファの値を入れるときに，すでにそこへ値があるかどうかで行います。

最近の3Dポリゴンゲームでは，数学的に物体が当たっているかどうかを判断していることが多いようです。どちらでもゲームに合った方法を選んでください。

Fig. 2H-6●パンチキーで始まる技のツリー構造。たとえば，「パンチ」「パンチ」「パンチ」と押されれば技③が，「パンチ」「キック」「パンチ」と押されれば技⑤が出る

サンプルゲームの遊び方

知っている人なら知っているであろうあの名作を実際に作ってみました。動きはさすがにあのとおりにはできませんでしたが，雰囲気はよく出ていると思います。

テンキーでキャラクタの操作，Z でキック，X でパンチというキーアサインになっています。DirectDraw を利用しているので，DirectX 7 以降がすでにインストールされている環境で実行してください。

グラフィックはポリゴンではなく2次元表示になっています。また，画面処理といった部分でのチューンはとくにしていないので，ちょっと遅いかもしれません。キャラクタの伸縮などはStretchBlt を使ってあらかじめパーツとして作ったものを利用しています。

表示関係はDirectDraw をコンポーネントとしてラッピングしたDirectCanvas クラスを使っています。キー関係は押されたキーに対応するフラグとしていったん格納しておき，それをタイマの処理がきたときに読み出しています。

List 2H-1 ●技データの格納（Delphi）

```
{ 型宣言 }
type
  TSkillItem = record                { 技の定義                }
    Num: Word;                       { 技の種類                }
    Damage: Word;                    { 相手に与えるダメージ    }
    Script: pTScriptItems;           { スクリプトへのポインタ  }
    Caption: string[20];             { 技の名前                }
  end;


{ 定数宣言 }
const
  CharacterPunch      = 0;
  CharacterKick       = 1;
  CharacterAskill     = 2;
  CharacterBskill     = 3;
  CharacterCskill     = 4;
  CharacterSkillLimit = 5;

  SkillItem: array[0..CharacterSkillLimit-1] of TSkillItem = (
    (Num: CharacterPunch;
      Damage: 2;  Script: @scrPunch;  Caption: 'ぱんち'),
    (Num: CharacterKick;
      Damage: 2;  Script: @scrKick;   Caption: 'きっく'),
    (Num: CharacterAskill;
```

▼

List 2H-1

```
      Damage: 5;  Script: @scrAskill; Caption: 'ばーすとぼいす'),
    (Num: CharacterBskill;
      Damage: 8;  Script: @scrBskill; Caption: 'めておふぉーる'),
    (Num: CharacterCskill;
      Damage: 10; Script: @scrCskill; Caption: 'がんばれおかのー')
  );
```

List 2H-2●技データの格納（C/C++）

```
/* 型宣言 */
typedef struct {              /* 技の定義                */
    word Num;                 /* 技の種類                */
    word Damage;              /* 相手に与えるダメージ    */
    pTScriptItems Script;     /* スクリプトへのポインタ  */
    char Caption[20];         /* 技の名前                */
} TSkillItem;

/* 定数宣言 */
#define CharacterPunch       0
#define CharacterKick        1
#define CharacterAskill      2
#define CharacterBskill      3
#define CharacterCskill      4
#define CharacterSkillLimit 5

static TSkillItem SkillItem[CharacterSkillLimit + 1] = {
    {CharacterPunch,   2, &scrPunch,   "ぱんち"},
    {CharacterKick,    2, &scrKick,    "きっく"},
    {CharacterAskill,  5, &scrAskill,  "ばーすとぼいす"},
    {CharacterBskill,  8, &scrBskill,  "めておふぉーる"},
    {CharacterCskill, 10, &scrCskill, "がんばれおかのー"},
};
```

Chapter 3

応用編 ゲーム作成の実例

最近のゲームでは既存のゲームの枠に収まらない新しい種類のゲームがぞくぞくと発売され，それがまたたいへんな人気を呼んでいます。こうしたゲームを実際に企画段階から作成し完成するまでを取りあげます。

Section 1 ゲームの題材を探す

この章では「ゲームになりそうな現実に起きているできごとや場面」を基にして，「それをどうゲームとして表現し，いかにして作るか」を解説することにします。ここまでに取りあげてきた「レシピ」はこのためにあったといっても過言ではありません。なぜなら現在考えられているコンピュータゲームのほとんどの処理は，過去に開発されたアルゴリズムを少しだけ発展させて再利用しているのです。まったく未知のゲームを作るときも，既存の手法が元となります。
すべての「レシピ」をふんだんに使い，「フルコース」としてみなさんに堪能していただくつもりです。

● ゲームを作ってみよう

あなたはどんなゲームを作りたいですか？　「みんなに遊んでもらえるゲーム」，「表現の場としてのゲーム」など，この答えは作る人によってまったく違います。納期前で何回朝日を見ても終わりが見えない泥沼状態のときは，この現実こそがゲームであってほしいと思うことがよくありますが，なかなかエンディングを迎えられません。どこかでフラグを立て忘れたのでしょうか？

毎年何千タイトルものゲームが，世界中で作られています。そしてそれらは国や人種を超え，さまざまな形でプレイヤーに遊ばれています。そのなかには大きな話題となるゲームもあるでしょうし，一方ですぐに忘れ去られてしまうものもあります。いったいどのようにしてゲームは産み出されているのでしょうか？

ここではこの疑問をできるだけ解決してみたいと思います。どのようにして新しいコンピュータゲームが作られるのか，その作業の流れを検証するために「まったく何もない状態からゲームを作る」ことにして，実例を示しながら解説します。読者の方々が実際にゲームを作る際の助けとなれば幸いです。

◆まずはテーマを選ぶ

いざ「ゲームを作る」というときにいちばんに決めるとしたら，「どんな内容のゲームにするのか」ということでしょう。具体的にどんなものをゲームにしたいのか，テーマとなるべきものをはじめに決めてしまいます。ここで決めたテーマを骨

格にすることで，これにさまざまな要素を肉付けし，ゲームとして完成させやすくなります。また，ゲームを作っているときに「こんなんでいいのか！」と思い始めたときも，このテーマに立ち戻ることで進むべき道が選びやすくなるでしょう。

ゲームのテーマはどこにでもあります。現実に起きているあらゆることを「ゲーム」にすることができます。世の中こそゲームそのものです。人の生い立ちや争い，経営が傾いた企業を立て直したり，レースカーや電車など一般にはなかなか操縦できないものでもゲームなら好きなときにいつでも運転できます。

逆にいまあるゲームを考えてみてください。格闘ゲームはケンカや武道の試合から派生したものですし，RPGに見られる剣と魔法の世界は，結局は神話と呼ばれる「かつてあった出来事」を元にしています。いっけん現実世界とは関係なさそうなパズルゲームも世の中の出来事を抽象化したものといえそうです。テトリスや倉庫番のように物をトランクや部屋にきれいに詰めるという作業は，日常でよくあるシチュエーションだと思います。

ちょっとしたアイデアでよいのです。これはおもしろいという1つのアイデアが生まれたら，それを伸ばしてみましょう。また，夢の中で得た不思議な体験を元にゲームを作ったという事例も実は多くあるようです。

ほかのゲームが新しいゲームの参考となることもよくあります。人間とは勝手なもので，お気に入りのゲームで遊んでいる最中に「ここの操作性が悪い」，「ここがこうなったら便利なのに」とか思うことがあります。それなら，いっそのこと，これを実現するようにゲームを作ってみるのもいいと思います。

きっかけとなるものは何でもいいのです。RPGやシューティングゲームなど，具体的な種類を決めてももちろんかまいません。「こういうのをゲームにしたいなあ」と感じられたら，それをだいじにしてください。

◆ゲームのテーマを決める

今回作るゲームのテーマを決めてみましょう。ひとことで言うと「いくつか感じていたことをまとめてみたらゲームになっちゃった」ということかもしれません。

だいぶ前にある方から「私たちの後輩が育たない。何事にも冷めている。技術が進み，宇宙などが身近になった分夢を見られなくなったからかもしれない」という話を伺いました。また，ある方は「アメリカのアポロ計画以降，さしたる技術革新がされていない。これでは技術者は何を夢にしたらいいのだ」という意見を述べていました。

こうした内容を聞いていて筆者がはたと感じたのは「宇宙が足りない！」とい

うことでした。21世紀を迎えたというのに宇宙開発はまだそれほど進んでいません。火星への植民もまだですし，月基地もできていません。ちょうど筆者自身も宇宙に飢えていたのだと思います。筆者が昔から感じていた「かっこよくて夢のある宇宙」を取り戻してみたくなったのです。これを今回のテーマにしてゲームを作成してみることにします。

● ゲームの対象を調査する

決めたテーマについてある程度理解していないと，なかなかおもしろいゲームが作れません。まず，テーマを決めたらそれがどのようなものなのか，確認を含めて考えてみることにします。

◆それはいったいどんなもの？

テーマの対象となるものを集めてイメージを固めます。そこからゲーム全体の仕組みを考えてみるとはっきりとした内容を得ることができます。

シミュレーションゲームなら実際にどのような動きをするのか調べてみるのもいいでしょうし，ファンタジーなど実世界とは関係ないものでは，どのような設定なのか，世界観を決めていくことになります。

筆者が考える宇宙は，どちらかというとSF書籍から得られた内容が多くを占めているようです。今回のゲームで真っ先に浮かんだのは『エンダーのゲーム』（オースン・スコット・カード著/野口幸夫訳，早川書房刊）でした。ある天才少年が対異星人戦争用指揮官を育てる学校に入り，さまざまな苦難を受けるのですが，そのなかで行うシミュレーションゲームがやたらとリアリティがあるもので感動したものです。ほかの宇宙観としてはC.J.チェリィやU.K.ルグィン，W.J.ウィリアムズ，笹本祐一……とまあ，偏っている気がします(笑)。ここまでを振り返ると戦艦を操って敵を撃破していくような戦略をシミュレーションするゲームがよさそうな感じです。

いろいろ考えていたところに，タイムリーな話題がありました。日本のすばる望遠鏡の観測によると全宇宙の2/3にあたる光源が銀河系から来たものではないという発表でした。そこで筆者は勝手に「銀河中心部で何者かが異世界の艦隊と交戦中で，そのときの爆発光なのだろう」と予測しました(笑)。こうしたニュースから世界観を創造していくのも楽しいものです。

◆どんなところが楽しいの？

イメージをふくらませたら，そこからゲームの本質となる「楽しいところ」を抽出していきます。シューティングゲームなどはその爽快感や追い詰められる感じが楽しいのでしょうし，ゲームによっては登場するキャラクタそのものがポイントとなるものもあります。その楽しさを見つけて，ゲームをデザインするときにそこをより伸ばしていくようにします。

戦略系のゲームとしては，わらわらといる味方の戦艦たちを操作して，敵を追い詰め，撃破するところにあります。この楽しさは将棋と同様のものだと思います。将棋というと「ついたて将棋」という遊び方をご存じですか？　将棋盤の真ん中についたてを起き，好きな駒を自陣に自由に配置します。並べたらついたてを外して普通に対局をします。このように敵の出方がわからないのもゲームのおもしろさを高める要素の1つになるはずです。

臨場感を増すように工夫していくのもゲームが楽しくなるポイントです。日本のSF映画やアニメでは，艦内にいるオペレータがダメージや命令を叫ぶ描写がよくあります。たとえば「第三艦橋大破！」とか「○○発進！」などです。ウィンドウを出してこうした叫んでいるシーンを表示するとおもしろくなりそうです。ほかにもダメージを受けると画面が赤くなるとか，ゲームを演出するときのネタとなるものをこの段階で集めておきます。

宇宙で楽しいところは，「未知の領域を開拓していく」ことです。宇宙は広大です。もし戦艦が実際に宇宙を航行しているのだとしたら，装備しているレーダーなどでは探査しきれないところが多くあるはずです。もし未知の領域へ踏み込んで何かいいことがあれば，ゲームとして楽しくなりそうです。戦略系のゲームであると同時にこうした宇宙を探検していく楽しさも含めたいところです。

◆反対の方向から考えてみる

自分が楽しいと思っていても，他人から見ればそんなに楽しいものではなかったということはよくあります。これはゲームでも同様です。もし，誰にでも楽しんでもらえるゲームを作りたいのなら，ときには「そのポイントはつまらないのではないか」，「本当に楽しいのだろうか？」といったように常に反対の立場に立って眺めることをしてみましょう。

たとえば「探検する楽しさ」は，あまりに多くなると冗長になりがちです。あるゲームでは草原を剣でなぎ払うとお金が出てくるそうで，草刈りゲームと呼ばれています。そこにはなぜ剣で草をなぎ払うことが本編のゲームと関係するのかとい

う必然性がありません。このままでは遊びではなく事務処理になってしまいます。また，敵の位置があまりにもわからないのでは，ゲームとはいえなくなってしまいます。いずれにしろ「やりすぎは禁物」です。

また，プレイヤーの心理を考えてみるのもいいと思います。「作り手の都合」という言葉がありますが，これはよくないことです。「プレイヤーをいかにして楽しませてあげるか」ということを考えたとき，何が楽しいかが必然的にわかってきます。また，どのようにしたらもっと楽しんでもらえるのかを考えることができます。ゲームの題材をもとにさまざまな要素を集めたら，これらについてプレイヤーの視点で考えながらもう一度見直してください。

集めた題材やゲームのポイントとなるところは，このような感じでざっとふるいにかけてからゲームそのものを考えていくようにすると，すっきりとしたよいゲームを作れることでしょう。

ゲームのために考えた要素を削ったり，伸ばしていくということは「ゲームデザイン」と呼ばれている作業の1つになります。では，具体的にはどのようにしてゲームをデザインしていくのでしょうか？

Section 2 ゲームをデザインする

Section 1で決めた題材からゲーム性を引き出して「コンピュータゲーム」として実現するために必要な作業が「ゲームデザイン」です。ここでは具体的にゲームのデザインを決めながら重要な点について述べていきます。

● ゲームの内容を決める

テーマを決めて，そこから楽しいと感じられるいくつかのポイントを見つけました。ここからゲームになりそうな内容を取り出して，1つのゲームとしてまとめていきます。これは，「国同士の戦い」があって，そこから「将棋」や「チェス」が生まれたり，前述のように荷物の詰め替えなどからパズルゲームが作られた関係と同じです。

作ったゲームがとてもおもしろくなるか，それともつまらなくなるかは，この段階で決まってしまいます。とはいえ，あとで何度か作り替えを迫られることがあり，柔軟にしておいたほうがいいこともあります。

ポイントとなるところをいくつか述べおきますが，これらはほんの一例にすぎません。いろいろと考えて工夫してみてください。

◆ゲーム性を見いだす

ゲームというのは「始まり」があって何かしらのルールによって行動をし，「勝ち負け」のある「終わり」を迎えることで成り立っています。レースゲームなどでは「試合」というゲームそのものをテーマにしているので勝ち負けの条件などもわかりやすいのですが，シミュレーション系のゲームではどんなことをすると勝ちとなるかでゲームの楽しさが決まってしまうようです。

もちろんすべてでこうなっているわけではありません。ですが，遊びとして成り立たせるには，このような流れであったほうがいいでしょう。決めたテーマの中からこうした流れが得られるように構成していくことにします。

◆切り口を定める

筆者がゲームを作るうえで重要だと思っているのは「ゲームの切り口」です。今

回のゲームは，戦略型のゲームだと考えていますが，ポイントを「敵の殲滅」に置くか，それとも「敵との交渉で領土を拡大していく」，「資源の調達や戦艦の製造に重点を置く」，「ストーリーがあってそれを読ましていく」など，1つのテーマでもいろいろなゲームが考えられます。これらをみんな取り込んでしまうのもいいのですが，それでは平凡なゲームとなってしまいます。

そこでどんな方向からのゲームにするのか，その切り口を考え，明確な「主題」を決めてしまいます。ここにいくつかの副題が主題を引き立てるように構成していくことで，遊んで楽しいゲームを作ることができます。あまりごちゃごちゃと要素を詰め込みすぎず，ゲームが持つシンプルな楽しさを引き出すように心がけましょう。

◆おもしろさのポイントを見抜く

前に決めたテーマの中から「楽しい」，「おもしろい」と思う部分をあげてみました。このおもしろいところをゲームの主題として，そこに味付けするようにゲームを構成していくと比較的楽に作れます。

また「もどかしさ」も重要です。簡単にはいかないように障害や制限を加えることで，その楽しさがより増してきます。たとえばシューティングゲームなどで障害物がなければ，あまり楽しくないはずです。こうしたものを考えて主題と副題の関係でゲームを構成するようにします。

◆欲求を満たすゲームを作ろう

ゲームは「遊び」です。遊びということは「どこかに楽しめる要素がある」ことになります。そうでなければ遊びではなくなってしまいます。この遊びはいろいろな「欲求を満たす」ものです。「笑い」，「さまざまな快楽」，「爽快感」など人間が心地よいと思うものを満たしてくれるのがゲームです。ゲーム中に「最後にこんなエンディングを得られたらいいなあ」といったプレイヤーの期待も欲求の1つでしょう。

この本質さえ押さえていれば「おもしろいゲーム」になります。つまらないといわれているゲームはいったいどこがよくないのでしょうか？　いろいろな理由があるとは思うのですが，いちばんの原因はこの本質をとらえられていないことです。「エンディングがつまらない」，「操作性が悪い」，「敵があまりにも難しすぎる」などはすべてこの心地よさを阻むものです。このようなマイナスとなる要素をできるだけなくし，ゲームやテーマが持つおもしろさのポイントを巧みにつければ，多くの

人に受け入れられるゲームとなります。

◆演出を考える

ゲームは現実をモデルにしているといっても，現実そのものではありません。より楽しさを得られるようにするため，いろいろな演出がされています。前述のように艦内のようすをゲーム中に表示させたり，実感的な画面デザインを考えてみるのも演出の1つです。といいつつ，実際にはスケジュールなどの都合でどんどん削られたり変更されてしまう部分かもしれません(笑)。

今回選んだテーマは，現実にあるものではありません。でも，「ある人たち」にとっては当たり前の世界です。グラフィックを担当しているMALUMI氏と企画について話しているとき，「高速戦艦といえばこれだよねー」，「すでに多くの世界観が出されているので，デザインがたいへんだよねー」といろいろな話題が出ました。こうした世界観を破壊してしまうのも楽しいことですが(個人的にはそのほうが好みなのですが……)，今回はできるだけゲームそのものが快適に楽しめることに重点を置いて演出面を抑えています。

あまり過剰な演出はゲーム本来の楽しさを損なってしまいます。処理に時間がかかるようなものでは，プレイヤーをゲームという仮想的な空間から現実へと引き戻してしまいます。どこまで演出を行うのか，その指針となるのは「自分がそのゲームをしていて楽しいと感じるか否か」ということです。作者が苦痛に思うことはきっとプレイヤーもよく思わないはずです。また，デバッグ中には何回もゲームをすることになるので，こうした苦痛に慣れてしまい問題点が見つけにくくなります。初めてプレイしたときに違和感を感じたら，そのままにしないで問題点を考えるようにしましょう。

◆ほかのゲームも参考にする

ユーザインタフェイスやゲームのシステムは，すでにあるゲームのほうが洗練されているはずです。逆にまったく新しい操作性を採用するとプレイヤーが混乱したり慣れるのに時間がかかってしまうことが考えられます。

プレイヤーがどういうふうに捉えるのかを推測し，ほかのゲームやさまざまなものを参考にしてもっともよいシステムを考えるようにします。

◆ゲームシステムを決める

こうしたポイントを考えて，ゲームシステムをまとめていきます。今回のゲーム

では，Fig. 3-1のようにしてみました。このゲームでは艦隊戦を中心にしたいと思います。実際に戦艦を作るとなると予算や材料，人員などが必要となりますが，今回は戦艦の指揮に注目することにし，純粋に「戦略ごっこ」が楽しめるようにしてみます。といいつつ，単純に自分がそうしたゲームで遊びたかったといのが最大の理由です(笑)。

ゲームは戦艦を何隻か選ぶところから始めます。戦略ゲームなので敵は明確に存在します。むしろ敵がいないとゲームとして成り立ちません。敵は異星人でも異次元人でもあるいは人類同士でもいいのですが，キャラクタとして味方と敵がはっきりとわかるようにしておきます。

ゲームとしては「自分の番」，「相手の番」といった「ターン制」をとります。カードゲームなどでおなじみのポピュラーな進め方です。できるだけオーソドックスなゲームシステムにして，ゲームそのものが楽しめるようにします。

次に楽しさの追求です。このゲームにはスターゲートがあります。スターゲートというのはSFによくでてくる仕組みの1つで，異空間をトンネルのように結んだその出入口のことをいいます。船が遠くの星へ行くとき，このスターゲートに入ることでいわゆるワープと同じ結果が得られます。ゲームでいえば，これは始めに戦艦が置けるところを制限するということになります。また，ゲーム中でも利用できるので，うまく活用すれば戦略の1つにできると思います。

マップは味方の戦艦がいる一定の広さの範囲しか表示されません。艦の種類によってこの範囲は変わります。こうすると敵がどこから現れるかわからず，敵を探

- ついたてを立てて適当に駒を並べるという対戦将棋の楽しさを持たせる
- 惑星からの緊急発進，要塞攻略のように戦略とそのシチュエーションの楽しさを得られるようにする
- ターン制
- ある一定の範囲に区切られた盤状のマップを使う
- マップは自動生成
- プレイヤは使いたい艦を選んでゲームを始める
- それぞれを操艦することでゲームを進める
- 敵の旗艦をやっつけるか全滅させたら勝利
- 画面は昔のダンジョンゲームなどのように奥行きがある視点で，ある程度奥は見せない
- 盤面はヘックス表示で，戦艦は浮いているように見せる
- マップ画面には戦艦を基準にして，ある一定の範囲しか表示しない
- 敵への距離に応じて攻撃力が増減する
- 移動ドックがある。移動ドックの周囲に僚艦を置くとダメージを回復できる
- スターゲートがある。敵艦も含めて出現できるのはスターゲートからのみとなる

Fig. 3-1 ●ゲームシステムの草案

す索敵という作業が重要になってきます。敵側にえらく性能のいい艦を与えるのも楽しいかもしれません。バランスを考えてこれらを調整していきます。

実際にプログラムにしてみるといくつか問題が起きてくるはずです。ユーザーが勝ちやすいなど条件が問題となることはよくあります。そのときはこの段階に戻って修正したあとで，それをプログラムへと反映していきます。こうしてゲームバランスを調整します。

さて，商品として販売するためのゲーム開発ではこれらに加えて「スポンサーを見つける」，「市場調査を行う」，「損益を考えて出荷数を決める」とお金に関係する内容もかかわってくるのですが，個人でのゲーム開発ではもちろん関係ありません。好きなようにゲームを作ってしまいましょう。

● 仕組みを決める

実際のプログラミングで使われるその仕組みについて，この段階でいろいろと検討し，方針を決めておきます。そうするとプログラミングを始めるときのとっかかりができます。

◆表示方法

シミュレーション系のゲームでは，物体をどの視点から見るのかによってさまざまな表示方法があります(Fig. 3-2)。今回はぷかぷかと船が浮かんでいるようすを見たかったので，Fig. 3-2-(c)のように奥行きがある状態とFig. 3-2-(d)のヘックス表示を組み合わせたような形にしました。

艦同士が奥に重なったような形になるときは，艦だけではなく，マップからも指定できるようにします。また艦の画像を盤の目よりも大きくして，クリックしやすいようにしておきます。

◆戦闘

もっとも単純な方法です。「攻撃するほうがサイコロを振る」，「受けるほうがサイコロを振り，その値が攻撃する値よりも大きければ防御成功で，そうでなければダメージを受ける」といった形になります。戦艦の種類によってそのサイコロの値が大きかったり小さかったりします。また，攻撃用と防御用のサイコロを変え，艦の性格がより出るようにします。

サイコロは乱数の仕組みがそのまま使えます。あとはそれぞれのやりとりをプロ

(a)平面　(b)斜め　(c)奥行き　(d)ヘックス(6角形)

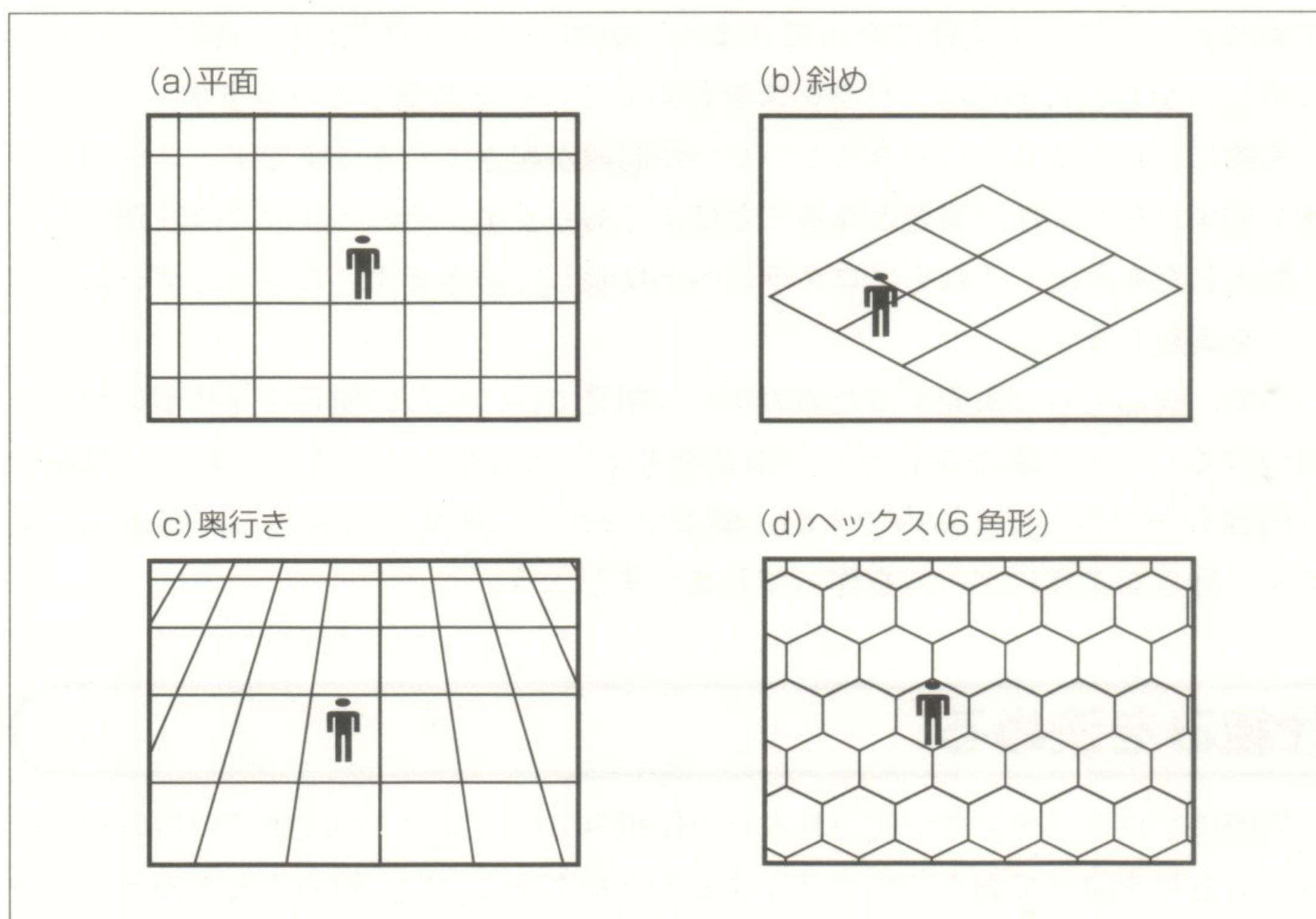



Fig. 3-2●ゲームの画面表示に使われる視点いろいろ

グラムとしてそのまま作ればいいでしょう。

シミュレーションということもあり，この戦闘の関係に「距離」を取り入れてみることにしました。敵との距離が遠くなるほど，攻撃する力が弱くなります。逆に近いほど最大の力を発揮します。どうしても破壊しなくてはいけないものの場合，「突撃して近くまで接近し，攻撃後に急速離脱する」といった戦略が使えることになります。

◆操作方法を考える

このようなゲームの場合，何度もプレイヤーが指示を行わなくてはならず，うまく操作方法を考えないと，遊ぶのがとても面倒なゲームとなってしまいます。

まず，「操作そのものを減らす」ことを考えます。ターン制ということで，艦隊をまとめて移動するとき，いちいち艦全部に移動を指定しなければなりません。そこで移動の場合，その移動先を何回かクリックすることで，その回数のターン分だけその方向に自動的に移動できるようにしました。こうしておくと索敵などでも便利に使えるでしょう。

このほかにも自動化できそうなところは自動化しておくなどいろいろな工夫を考

えておきます。ある程度ゲームができてから実際にプレイし，めんどうに思ったところを改善していく方法がもっとも現実的かもしれません。

◆仕様の完成

これでゲームの仕様が完成しました。これらを企画書などにまとめてわかりやすくしておきます。もしプログラミング中にどう作っていくべきか迷ったときは，この企画書に戻って検討することにします。

なお，ゲームが動作する環境はWindowsでDirectXを使わないことにしました。単に自分のノートパソコンでゲームがしたかっただけです(笑)。3D化してもおもしろいでしょうし，それはゲームのデザイン次第だと思います。

Section
③ ゲームをプログラミングする

ゲームというプログラムは，ちょっとしたものでも多くの工程を必要とします。とくにいま作ろうとしているゲームは，ほかのゲームと比べると未知の部分が多くあります。ですが基本的にはどんなゲームでも「全体像」は同じです。まずはこれを考えることから始めます。

● ゲームの全体像

ゲームの全体像を検証してみましょう。そもそもゲームプログラミングとはいったいどういうものなのでしょうか？　よくある答えとしては「タスク管理」，「ループ」，「状態を次へ変えていく状態遷移」などがあると思います。これらはすべて正解です。なぜなら全部使わないとゲームの処理が成り立たないからです。

ここまで何度もやってきたことですが，プログラムを作りたいときは，すでにあるプログラムからその仕組みや処理の流れを推理して，それをとっかかりにして作っていくとうまくいくものです。

実際のゲームをプレイするところを思い出してください。ゲームの種類は何でもかまいません。まずゲームを起動すると，ゲームのスタートや設定を変えられる項目「メニュー」が表示されます。しばらくそのままでいるとゲームの「オープニング」が表示され，それが終わると再度メニューに戻ります。メニューからゲームのスタートを選択すると，実際のゲームが開始されます。ゲーム中に勝ったり負けたりすると勝負の結果などが表示されて，再度メニュー画面が表示されます。表面的に見える処理の流れはFig. 3-3に見られるような形になっています。

◆画面を定期的に表示する

これをさらに解体してみます。ゲーム画面にはさまざまなキャラクタが表示されています。ほとんどのゲームではこれらが動いているように見えています。

この仕組みはアニメーションと同じです。セルや紙に1コマずつわずかにずらしてキャラクタを描き，これをパラパラとめくっていくことで動いているように見えます。ただ，ゲームではプレイヤーの操作に応じてキャラクタを変えたり，データの量に制約があるので，このままの方法は使えません。

ゲーム起動
メニュー
オープニング
設定
設定モード
スタート
ゲーム中
終了条件に達した？
それならゲームが終了
終了
ゲーム終了

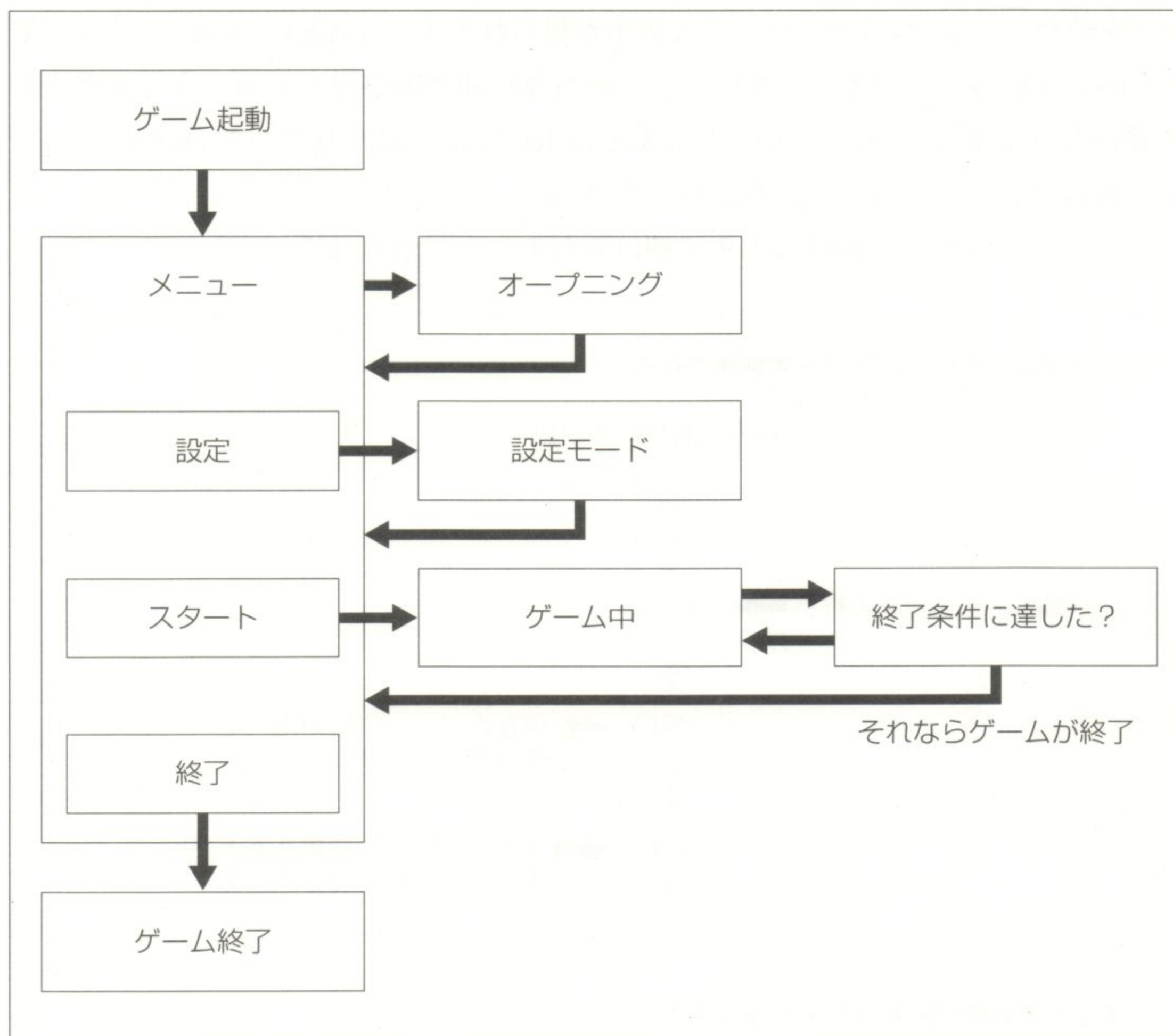



Fig. 3-3●表面的に見えるゲームの流れ

そこで「画面の組み立て」を行います。まず，画面サイズ分の背景となるものを用意しておきます。そこにキャラクタを重ね合わせて，この背景を定期的に呼び出し，そのたびにキャラクタの位置や絵を変えれば，動いているように見せることができます。

定期的に画面を表示するにはwhile文などのループを使って画面表示を行うコードを呼び出していけばいいのですが，「定期的」というところがポイントです。先ほどのパラパラアニメの場合，1コマずつの時間がずれてしまうと，とたんに動きがぎくしゃくしたものになってしまいます。そこで表示する時間間隔を決め，決まった時間ごとに画像を変えるようにします。この「ある決めた時間」を超えて処理が行われるのが，いわゆる「処理落ち」です。

また，時間間隔は通常ディスプレイの垂直同期信号(VSYNC)の間隔と同じ時間が使われます。ディスプレイも定期的に画面を書き換えているので，これとタイミ

ングを合わせることでよりスムーズな表示が得られます。WindowsではこのVSYNCがDirectX以外では利用できないので，適当な時間間隔が得られるように調整する必要があります(これらについては付録CD-ROMに収録されているOnIdleとスレッドの各サンプルプログラムをご覧ください)。

このループはゲームが終わるまで続けられることになります。

○場面の移り変わりをフェーズ処理で行う

現在の場面を示す変数

2番目

↓

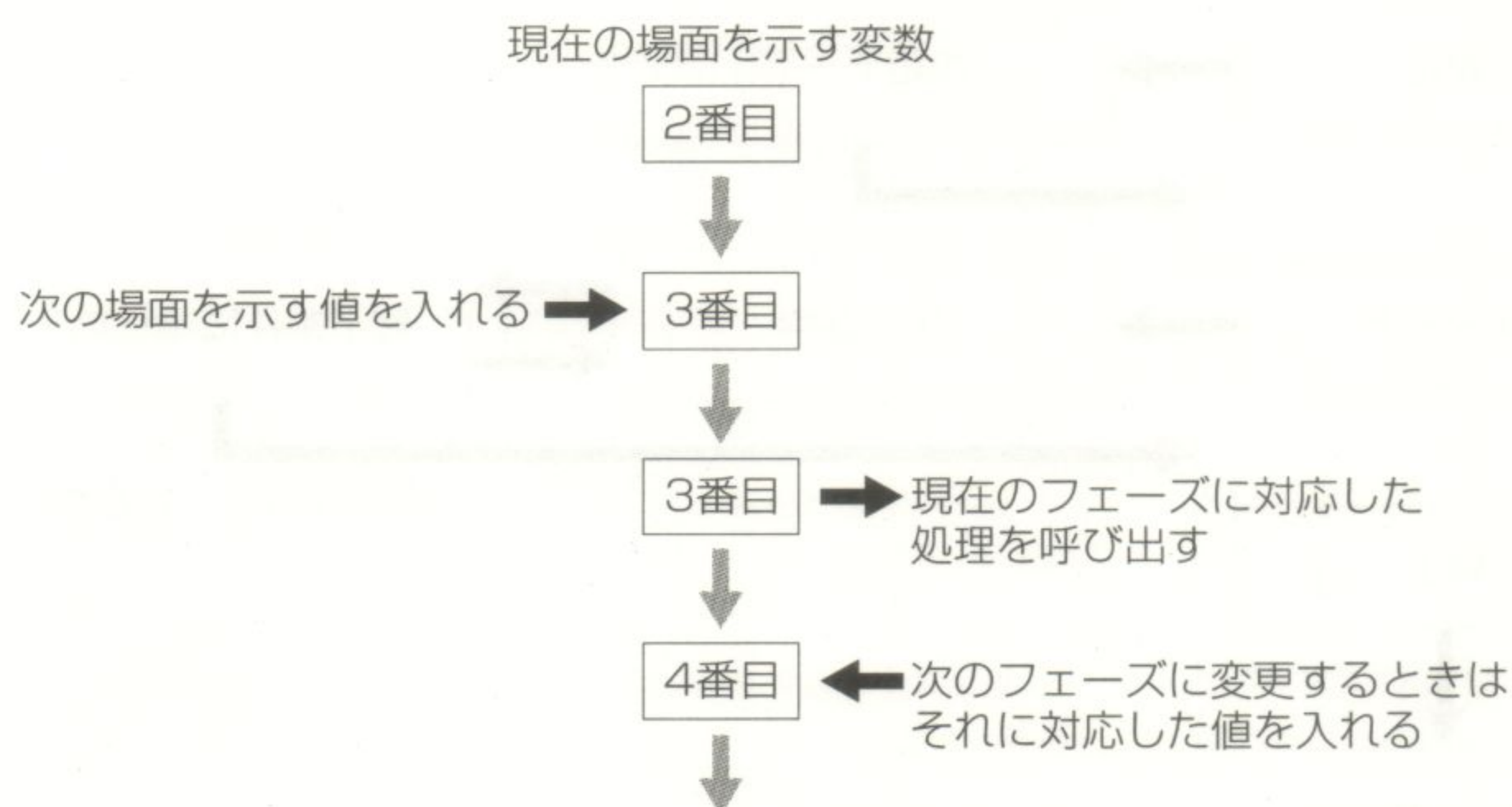
次の場面を示す値を入れる → 3番目

↓

3番目 → 現在のフェーズに対応した処理を呼び出す

↓

4番目 ← 次のフェーズに変更するときはそれに対応した値を入れる

↓

○フェーズの移り変わりをデータにする

現在の場面	対応する処理	次の場面
敵の番	敵を表示する	敵の思考
敵の思考	思考を行う	行動
行動	各艦に行動を指示	攻撃

⋮

○データを実行するルーチン

```
データが終わるまでループ
    現在の場面の値は“データ[カウンタ].現在の場面”と同じかどうか
        それなら“データ[カウンタ].対応する処理”を実行
        現在の場面 = データ[カウンタ].次の場面
    end;
end;
```

Fig. 3-4●フェーズ処理

◆さまざまなキャラクタの管理

ゲームの画面には複数のキャラクタが表示されます。これらを管理するには，まずアニメーションの位置などの必要な値をキャラクタごとに確保し，この値に対応するように画面へキャラクタを描画します。そして，ループ1回分の間に，表示するキャラクタの数だけこの処理を行います。表示するキャラクタの登録/消去は専用のテーブルを使って管理します。

これをある種のタスク処理のようにして制御しているという方もいます。またスレッドを使った管理もできますが，このような形が一般的のようです。

◆フェーズ処理

ゲームにはいろいろな段階や場面があります。ゲーム終了となることもあれば，キャラクタを選択するときもあるでしょう。これを場面ごとに条件分岐で行うと，場面を追加したくなったときに何かと面倒な作業が増えてしまいます。

そこで状態遷移やフェーズ処理などと呼ばれる処理が利用されます(Fig. 3-4)。ある1つの変数を用意しておき，その変数が現在の場面を示すとします。次の場面へ変更する操作はこの変数を書き換えるだけです。場面ごとの処理では，この変数を調べることで自分が対応するべき場面になったかどうかをチェックできます。

さらに「現在の場面」，「対応する処理」，「次の場面」といった3つのデータを持つ構造体を用意し，これを配列にしておくと容易に場面の流れを記述できます。この方法の利点は場面の流れが変更しやすいことです。条件によって次の場面を変えるような処理にも簡単に対応できます。

◆新しいゲームでも根本は同じ

ゲームの流れをまとめるとFig. 3-5のようになることがわかったと思います。プログラムとはその名の通り処理や作業の流れを示すものです。この流れをプログラムとして作っていきます。逆にいうと「ゲーム終了まで続けられるループ」，「画面表示」，「表示などを定期的に行う方法」，「キャラクタの管理」，「プレイヤーからの入力」といったこれらの要素がそろっていれば，どんな環境でもゲームプログラムを作れるということになります。

新しいゲームを作るときもこの仕組みはまったく変わりません。この方法をそのまま使って，ここにゲームの処理をいろいろと加えていくことでプログラミングを行っていきます。

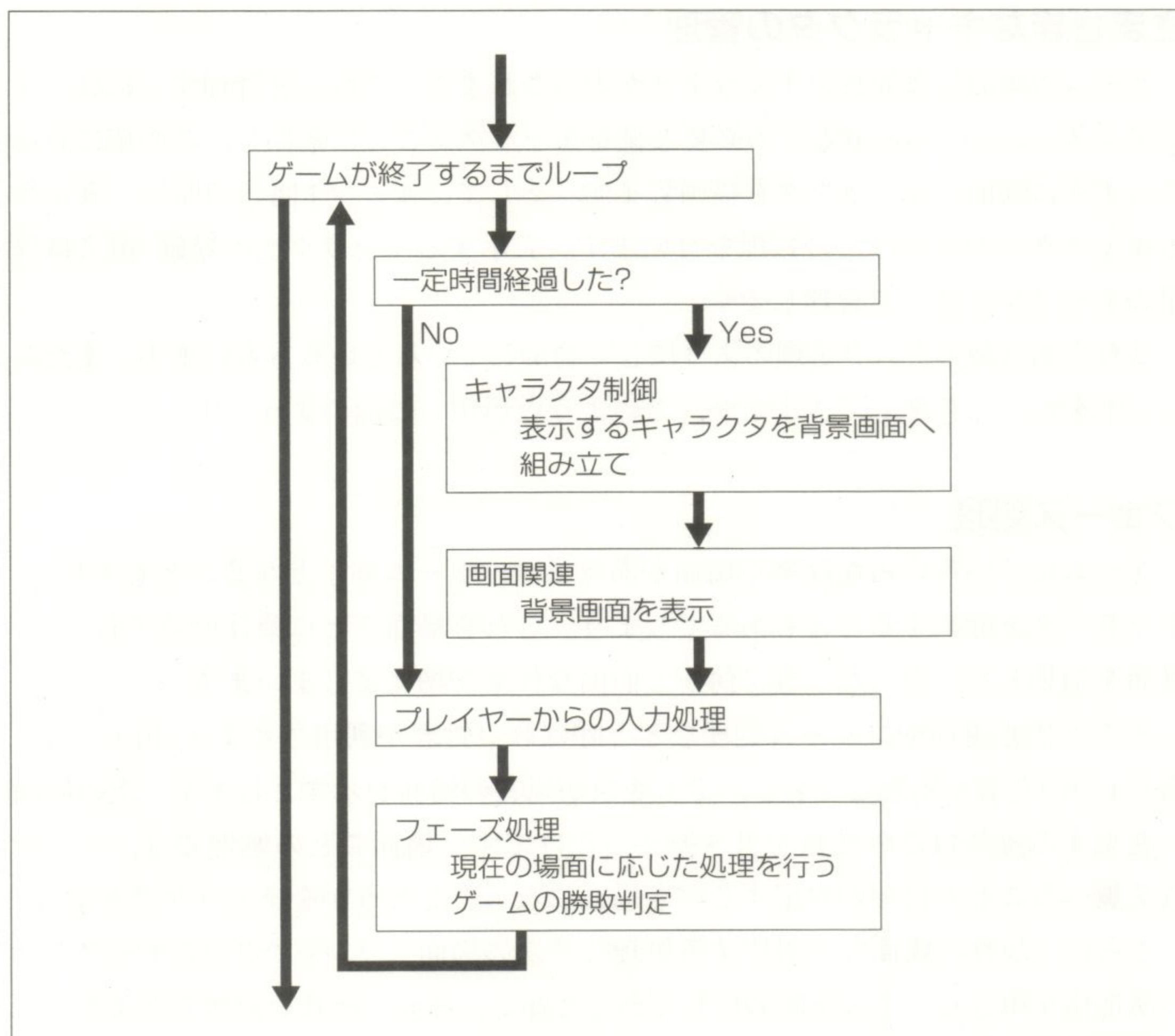



Fig. 3-5●ゲーム中の内部の流れ

● アルゴリズムの組み立て

それでは実際にゲームを作っていきましょう。全体的な構成は前述した方法そのままなので，今度はそれらが呼び出す処理について解説していきます。

◆マップ作成と管理

マップの作成は，迷路のときと同じようにランダムに障害物を配置していきます。ここではごく簡単に2次元配列をクリアして，障害物のキャラクタを示す値を入れていくようにしています。この方法だと有限の宇宙となってしまいますが，とりあえずこれでよしとします。

戦艦などのキャラクタを表示するときは，このマップにそれを示す値を与えま

す。これらはRPGなどと同じ考え方です。

◆マップ表示

ヘックス表示とはいえ，ただの2次元配列として扱えるように工夫します。連結している方向はどうでもいいのですが，そうではない方向のときは，Fig. 3-6のようにジグザグにたどるように扱うことにします。

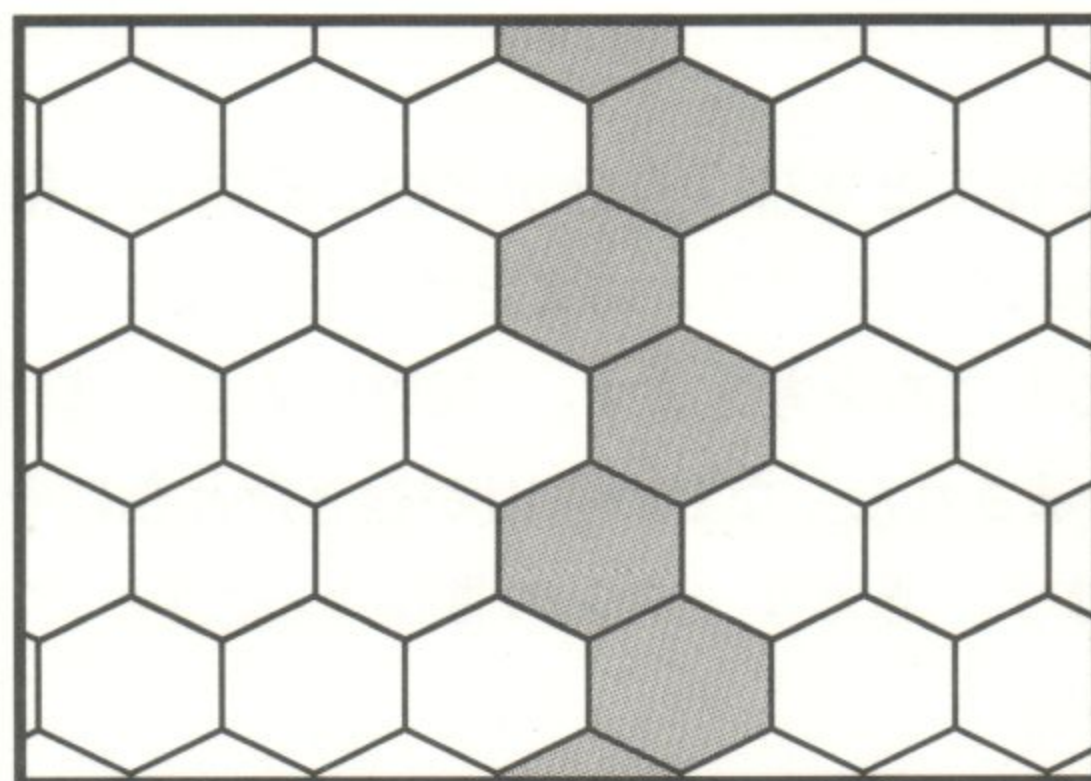
戦艦の操作のときに，その戦艦がいる位置から3つ左にある位置に何がいるか確認したいということがあるとします。そのときは，「方向から相対座標に変換するテーブル」を用意し，現在のマップの位置にこのテーブルの値を足すことで，調べたいマップの位置を求めます。こうしておくと，先ほどのジグザグ方向にも対応しやすくなります。範囲などもこのような形で処理します。

Mapdata[x][y](C/C++)，Mapdata[x, y](ObjectPascal)
としたとき，グレーの部分がy方向の値となる

Fig. 3-6●連結されていない方向の配列との対応

◆戦艦/キャラクタ管理

通常のキャラクタ管理と同じように，「キャラクタ1つ当たりに必要な値」をまとめた構造体を用意します。この値には対応する戦艦の画像，どこまでアニメーションが進んだのか，爆発しているのか，艦の性能などといったものが含まれます。

この構造体を艦の数だけ用意し，配列として扱います。

◆戦艦/キャラクタの表示

本書でも何度か取りあげているように画像の重ね合わせ処理でマップ画面と合成します。マップの位置に対応して，実際の画面のどの座標に表示させるかは，テーブルを使って変換するようにします。

◆戦艦/キャラクタの選択

マウスをクリックしたとき，その位置が戦艦かマップなのかを知る必要があります。また，戦艦の画像が重なることも考えなければいけません。

そこで画面サイズ分だけの画像をチェック用として別に用意しておき，クリックされた座標にあるチェック用画像の色を取り出して，これを識別するようにするというややぜいたく(？)な仕組みを用意しました。

もちろん座標を計算でチェックするといったほかの方法でもいいでしょう。

◆敵思考ルーチン

敵の思考ルーチンを作るときは「自分がどのようにして相手と戦うか」ということを考えていき，それをプログラムにまとめていきます。

まず「意思」となるところを考えます。「反撃する」，「逃げる」という行動を示す変数を用意し，「攻撃された」などの原因が起きたときにそれに応じて行動を示す変数の値を増減します。思考ルーチンの番になったら，この行動の値に応じて僚艦の行動を決めます。

◆ターン処理

ターン処理は前述のようにフェーズ処理としてまとめておきます。「自分の番」，「相手の番」，「行動」を示す値を用意し，それぞれに応じた処理を用意します。実際にはゲームの処理に応じてさらに多くの値を定義することになります。

このような処理を作るときは，ただたんに「自分の番」を示す値を1つ用意するのではなく，「処理に入る前」「処理」「処理が終わった」というようにしておくと，あとで変更しなければならないときに対応が楽になります。

◆処理をまとめる

これらの処理をまとめることでプログラムの完成となります。実際にはそれほど変わったアルゴリズムは使用していないことがわかるでしょうか？　仕組みを理解していれば，その仕組みを組み立て直したり，少し変えることで，いつでも新し

いゲームを作ることができます。

サンプルプログラムで確認する

作成されたサンプルプログラム「宇宙が足りない！」は，本書のサポートWebページ「http://cmaga.zdnet.co.jp/books/recipe/」で公開されています(Fig. 3-7)。

遊び方は次のような流れになります。最初に派遣する戦艦の種類とその数，どれを旗艦にするかを決めます。次に味方の戦艦を出現させるスターゲートを選択します。このあとでゲームが開始されます。

ターン制なので，プレイヤーの番になったらマウスで戦艦を選んで「移動」，「攻撃」などの指示を与えます。移動先や攻撃する戦艦は画面上に指示が出るのでそれをクリックします。ちなみに味方の艦も攻撃できてしまいまうので，気をつけてください。

こうしてゲームを進めます。敵を殲滅するか旗艦を破壊すると勝ちとなります。逆に自分の味方が全滅するか旗艦が破壊されると負けです。

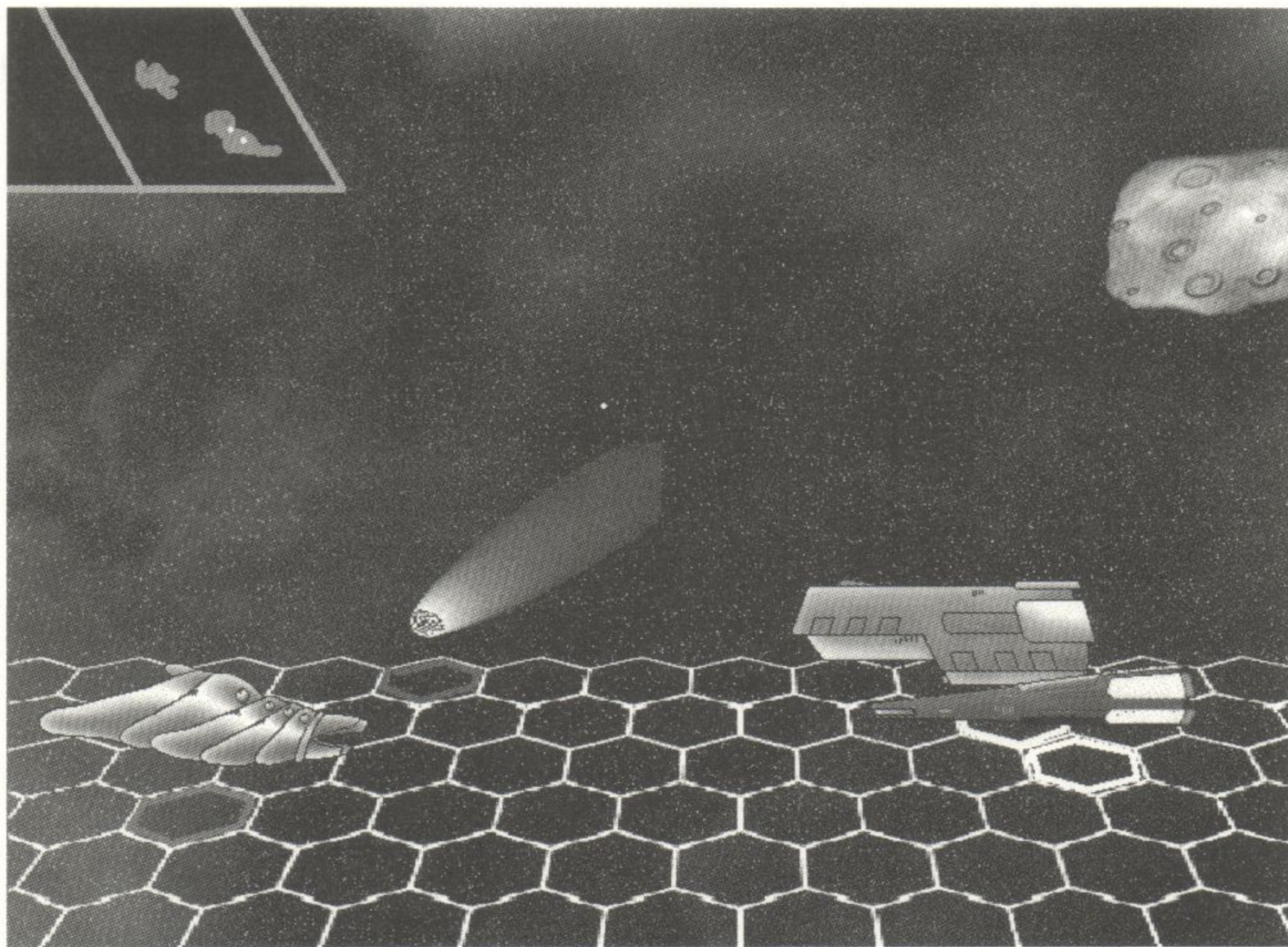



Fig. 3-7 ●宇宙が足りない！

マップは味方の艦船がいるところのみ表示されるので，あやしいところはとにかく偵察艦を派遣させてから全体を移動させるといいでしょう。またごくたまに彗星に出会います。彗星の進路上に戦艦があるとぶつかってしまうので，回避するようにしてください。

◆ゲームは作るのが楽しい

ソースコードに手を加えることで自分の宇宙を作ることもできます。いろいろと処理を追加して遊んでみてください。筆者がふと思うアイデアですが，宇宙という舞台を用意し，そこでさまざまなゲームができるようにしてしまうのも楽しいかと思います。たとえばあるプレイヤーが海賊(宙賊？)になって宇宙の覇権を握ろうとするのなら，もう片方では体制側になって海賊を取り締まることを目的としたり，また別のプレイヤーは流通や経済でこれらをコントロールするというように，パラレルにさまざまなゲームが影響を与えていく仕組みを作ると楽しそうです。どなたかやりませんか(笑)。

ゲームというのは切り口が変わればそれだけのゲームができてしまいます。でも，基本的な仕組みは変わりません。あることをゲームにしたいとき，その処理の内容や流れを考えていくことで，実際にプログラムとして組み立てていくことができるようになります。これはゲームをただ遊ぶ以上に楽しいこととなるでしょう。

Chapter 4

C++BuilderとDelphiでのプログラミング

本書で紹介したサンプルゲームはすべてBorland社のC++ BuilderとDelphiという開発ツールで作られています。これらのツールには，ゲーム作りの工程を簡略化してくれる便利な機能が多数搭載されています。この章ではC++BuilderとDelphiの機能の活用法を紹介します。

Section 1 C++Builder & Delphi とは

「C++Builder」と「Delphi」は，Borland 社から発売されている，Windows プログラムを作成するための開発ツールです。言語としては，C++Builer は C++ が，Delphi は Pascal をオブジェクト指向にした ObjectPascal が採用されています。C++Builder と Delphi とはどのような開発ツールなのでしょうか。

● 先進的な RAD ツール

C++Builder と Delphi でのプログラミングは，いずれも「フォームと呼ばれるウィンドウへ，ボタンやボックスなどといったコンポーネントを貼り付けていく」ことで1つのプログラムを作りあげていきます。「ボタンが押された」ときに「ウィンドウを出す」などの処理は，それに対応する「イベント」を加える作業をすることになります。そのイベントに反応する関数の外枠を自動的に生成してくれるので，プログラマはイベントに対する処理だけを記述すればよい仕組みになっています。

このような「外見から作り，それに対するイベントを記述していく」作成ツールは「RAD(Rapid Apprication Development)」と呼ばれています。C++Builder や Delphi 以外には，同社の JBuilder，Microsoft 社の Visual Basic などが有名です。

◆独自のクラスライブラリを搭載

構造的にはC++Builder/Delphi の両方とも「VCL(Visual Component Library)」と呼ばれるクラスライブラリで構築されています。このなかには，Windows において基本的な「ウィンドウ」，「ボタン」，「グラフィック」などの機能が含まれています。C++Builder と Delphi の両方で，同じ Object Pascal で記述されたクラスライブラリを使っているので，言語の違いこそありますが，クラスやコンポーネントの扱いは同じです。

一方で，「RTL(RunTime Library)」というものもあります。こちらにはそれぞれの言語に特有な「printf」，「write」などの機能が含まれています。これに加えて，Windows を操作するために使う Windows API の機能が使えます。これらはすべて混ぜて使うことができます。

Windows標準のGUIを簡単に構築

ではゲームをプログラミングする際に，C++BuilderとDelphiをどのように使えばよいのでしょうか？

Windows用のゲームを作る場合には，ボタンなどのWindows側が提供している機能をユーザインタフェイスとして利用する方法と，DOSのゲームのようにほとんどのユーザインタフェイスをゲーム側で用意するものの2種類があります。前者ならVCLを使えば「Windowsのシステムを操作するのに必要な」ほとんどの機能をネイティブに作るよりも簡単に作ることができます。とくにグラフィック関連のVCLはかなり強力です。ゲームでは，基本的に必要なアニメーションパターンなどはあらかじめ用意しておくので，描画のための機能はそれほど多く使いませんが，VCLは必要にして十分すぎるほどの環境を与えてくれます。後者でもWindowsのプログラムである以上，プログラム起動に必要な手続きが省けるので，ある程度楽ができます。

サンプルゲームで確認する

付録CD-ROMに収録されたサンプルゲームは，『Inside Windows』連載時にDelphi 2.0（一部Delphi 1.0）で作ったゲームを一部修正し，C++Builder 5，Delphi 5に移植したものです。これらのサンプルゲームをC++Builder 5やDelphi 5に読み込んで，各コンポーネントがどのような形で使われているのかを確認してみてください。

ゲーム用に作成したコンポーネントは非ビジュアルコンポーネントとなっているので，コンポーネントパレットには含めず，［ファイル(F)｜ユニットを使う(I)］などでコンポーネントのファイルを指定して，ユニットとして使ってください。

Section 2 C++Builder & Delphi のプログラミング技法

C++BuilderとDelphi でのゲームプログラミング技法についていくつか触れてみましょう。C++Builder や Delphi には，プログラミングに便利なコンポーネントが多数搭載されています。それらを活用することで，ゲーム作りに付きものの手間を省くことができます。なお，本文中でとくに記述がない場合は，C++Builder と Delphi の操作は共通となります。

初期化の処理

ほとんどのWindowsゲームは，Fig. 4-1のような処理の流れになっています。C++ BuilderやDelphiのゲームプログラムでは，初期化を「ゲームのための初期化」と「フォームやコンポーネントに関する初期化」の2つに分けなければなりません。

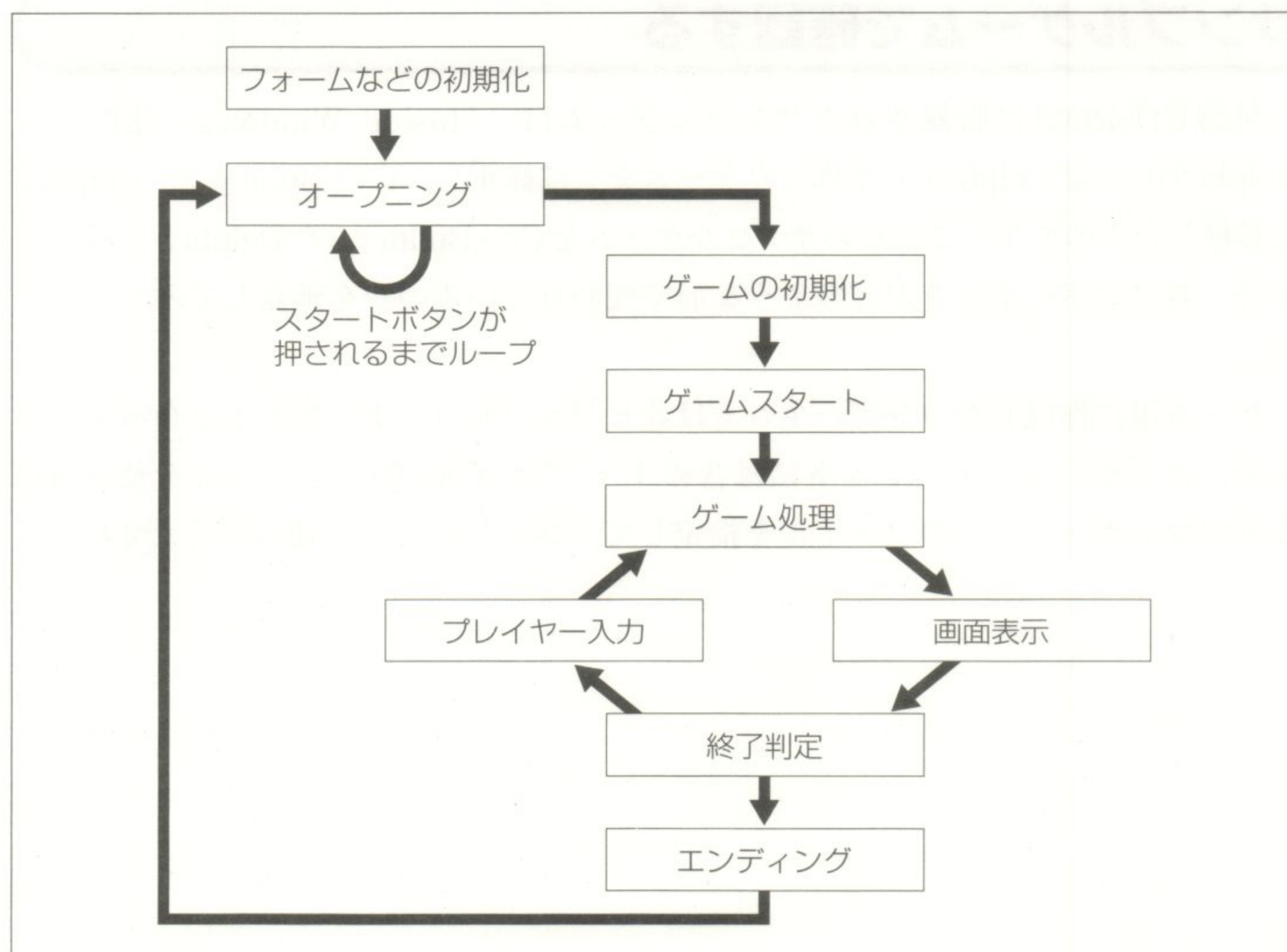



Fig. 4-1 ● Windows ゲームの全体の処理の流れ

◆ゲームのための初期化

ゲームのための初期化とは「キャラクタパラメータの初期化」「スクロール位置の設定」など「ゲームに関するすべて」を初期化するものです。これは別関数(メソッド)として作ります。ゲームを始めるときはもちろんですが，ゲームオーバなどでリプレイするときにもこの初期化関数を呼ぶことになるので，フォームの位置や形など「何度もやるとムダになってしまうもの」，メモリの確保など「何度もやると問題になるもの」はここでは初期化してはいけません。

◆フォームやコンポーネントに関する初期化

ゲームの初期化関数でできないものは，すべて「フォームやコンポーネントに関する初期化」で行います。これはメインフォームの「OnCreateイベント」に初期化用のコードを記述するだけです。そのフォームに固有の「Captionの設定」「メニューの設定」といったものは，そのフォームごとにそれぞれのOnCreateイベントを作って処理します。ただし，この段階ではコンポーネントのSetFocusメソッドなどが呼び出せません。そのときはこれだけ別関数としてまとめて，プロジェクトファイルからこれらを呼び出すようにします(List 4-1，4-2)。

ゲームの流れによっては，この別関数やOnCreateイベントから，ゲーム初期化用関数を呼び出してもかまいません。

List 4-1 ●初期化処理(Delphi)

```
{プロジェクトファイル}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Form1.Init;  // 初期化関数
  Application.Run;
end.

{ユニットファイル}

{プロジェクトファイルから呼ばれる初期化}
procedure TForm1.Init;
begin
    GameOpening;      {オープニングの実行}
    SetFocus;         {フォーカスの設定}
end;

{ゲームの初期化}
procedure TForm1.Init;
```

List 4-1

```
begin
    {パラメータなどゲーム処理に必要な初期化}
end;

{フォームでの初期化}
procedure TForm1.FormCreate(Sender: TObject);
begin
    {フォームやボタンに関係する初期化}
end;
```

List 4-2●初期化処理(C++Builder)

```
/* 初期化 (C++Builder) */

//プロジェクトファイル
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Form1->Init();  // 初期化関数
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}

// ユニットファイル

// プロジェクトファイルから呼ばれる初期化
void __fastcall TForm1::Init(void)
{
    GameOpening();  // オープニングの実行
    SetFocus();     // フォーカスの設定
}

// ゲームの初期化
void __fastcall TForm1::GameInit(void)
{
    // パラメータなどゲーム処理に必要な初期化
}

// フォームでの初期化
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    // フォームやボタンに関係する初期化
}
```

● ファイル読み込みとメモリ割り当て

ゲームプログラムでは「グラフィックデータ」「Waveデータ」など，非常に多くのファイルを扱います。C++BuilderとDelphiには，これらのファイルの表示や再生のための専用コンポーネントが用意されているので，それに付属の「LoadFrom File」メソッドなどを利用すれば，プログラムに読み込むことができます。

◆ユーザが定義したデータの読み込み

一方，「マップデータ」「シナリオデータ」などのみずから定義したデータにはクラスが用意されていないので，そのままではメモリに読み込めません。これらのデータを読み込むための処理は，プログラマ自身が作ることになります。

DelphiとC++Builerでは，それぞれRTLやAPIでファイル/メモリを操作する機能がありますが，「TMemoryStream」クラスを利用するのが何かと便利です。

使い方はコンポーネントと同じように，

```
C++Builder    memblock = new TMemoryStream;
Delphi        memblock := TMemoryStream.Create;
```

として実体を割り当てたあとに，

```
C++Builder    memblock->SetSize(メモリサイズ);
Delphi        memblock.SetSize(メモリサイズ);
```

として，使いたいサイズのメモリを割り当てます。ファイルの読み込みには「Load FromFile」，書き込みには「SaveToFile」の各メソッドが利用できます。

確保したメモリにアクセスするには，「Writeで書き込み」「Readで読み込み」することができます。またMemoryプロパティは汎用ポインタ型なので，

```
C++Builder    p = (TTestStruct *)memblock->Memory;
Delphi        p := PTTestStruct(memblock->Memory);
```

のようにキャストして使ってもよいでしょう。

これらの参照位置を変更するときは「Seek」メソッドを使います。なおMemoryプロパティは読み取り専用のプロパティのため，直接操作することはできません。キャストして設定したポインタ，前述の例なら「p」のほうを操作します。

マップデータの管理

マップデータは基本的に「2次元配列」となります。この大きさの分だけ確保したTMemoryStreamクラスのブロックに読み込んだり，TMemoryStreamを継承したクラスとして生成します。

まずマップの大きさを決めてしまいます。ロールプレイングゲームならキャラクタが1回に歩く移動量1つを基準にして，マップの大きさを決めます。これを四角形の「マス」として考えます。1マスごとに「地面」「家」などの情報が含まれることになります。

このマスを縦500×横500個だけ平面に集めると，これがマップデータ全体の大きさとなります。この平面1マスに対応する情報を1バイトとして考えれば，「500 × 500 = 250,000バイト」もの大きさになります(Fig. 4-2)。

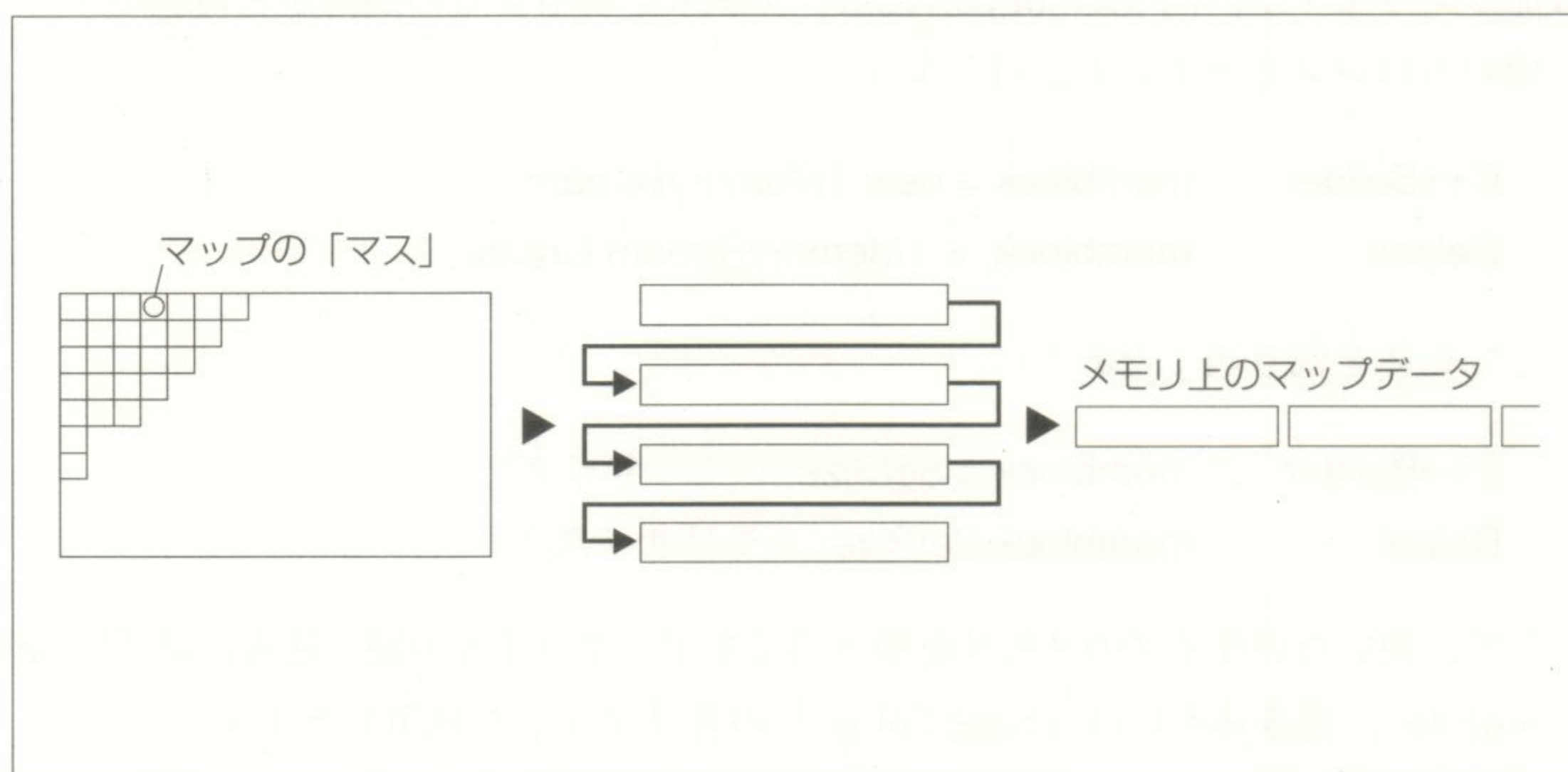



Fig. 4-2 ●マップを「マス」単位の2次元配列にして，配列の横1列ずつ順番にメモリに格納していく

◆マップデータファイルの読み込み

マップデータをファイルとして管理するのなら，それをプログラムへ読み込まなければなりません。これには，TMemoryStreamクラスのLoadFormFileメソッドを使えば簡単です。

マップのある地点の情報を取り出したいときは，2次元の座標からマップデータが格納されているメモリの位置が必要です。もし横方向のマス数が決まっているのなら，これをXの最大数として，

メモリの位置 ＝ (Xの最大数×Y)＋X;

という計算式が使えます。これをSeekメソッドで，

C++Builder　memblock->Seek((X_MAX * dy) + dx, soFromBeginning);
Delphi　memblock.Seek((X_MAX * dy) + dx, soFromBeginning);

として位置を移動させてから，値を取り出します(List 4-3，4-4)。

List 4-3●2次元マップデータの操作(Delphi)

```
{ Mapdata に代入 }
procedure TMap.SetPoint(X, Y: Integer; Data: byte);
var
  ofs: word;
begin
    if (X < FWidth) and (Y < FHeight) then begin
        FMapdata.Seek(FHeight * Y) + X, soFromBeginning);
        FMapdata.Write(Data, Sizeof(Data));
    end;
end;

{ Mapdata から値を取り出す }
function TMap.GetPoint(X, Y: Integer): byte;
var
  ofs: word;
  Data: byte;
begin
    if (X < FWidth) and (Y < FHeight) then begin
        FMapdata.Seek(FHeight * Y) + X, soFromBeginning);
        FMapdata.Read(Data, Sizeof(Data));
        result := Data;
    end else begin
        result := 0;
    end;
end;
```

List 4-4●2次元マップデータの操作(C++Builder)

```
// Mapdata に代入
void __fastcall TMap::SetPoint(int X, int Y, BYTE Data)
{
    WORD ofs;

    if ((X < FWidth) && (Y < FHeight)) {
        FMapdata->Seek(FHeight * Y) + X, soFromBeginning);
        FMapdata->Write(Data, Sizeof(Data));
```

▼

List 4-4

```
    }
}

// Mapdata から値を取り出す
BYTE __fastcall TMap::GetPoint(int X, int Y)
{
    WORD ofs;
    BYTE Data;

    if ((X < FWidth) && (Y < FHeight)) {
        FMapdata->Seek(FHeight * Y) + X, soFromBeginning);
        FMapdata->Read(Data, Sizeof(Data));
        return Data;
    }
    return 0;
}
```

◆マップデータから値を得る

あとはゲームによって取り扱い方が違ってくるので，とりあえず値を取り出すところまでをクラスとして作り，そこから先の「マップデータの値からテーブルなどで別の情報を取り出す」といった処理はオーバライドした関数が独自に処理するようにします。たとえば，

GetMapdata(dx,dy);

というもともとのメソッドがあって，ロールプレイングゲームのようにそこが地面なのか通行可能かどうかのフラグを得たいとします。マップを管理する基本クラスと，それを継承した新しいクラスを作って，そこで，

GetMapdata(dx,dy,ParamStruct);

としてオーバライドします。このメソッドの中で元のクラスのメソッドからマップデータを取り出して，そこからテーブルなどで得た必要な情報を構造体へ返す形にします。

この基本となるマップクラスそのものに手を入れてもよいのですが，そうするとほかのゲームで使うときにいらない機能を削ったりしなければならず，結局はめんどうになってしまいます。

画面表示のちらつきを抑える

画面を更新したときのちらつきを抑えるのが，画面表示処理での最大のポイントです。

まず仕組みとして，「フォームにTImageコンポーネントを表示するサイズだけ貼り付ける」と「貼ったTImageコンポーネントと同じサイズのTBitmapオブジェクトを用意する」の2つがあります。

ゲームでは，スプライトなどで背景とパーツを1つの画面へ組み立てなければなりません。これらは用意したTBitmapオブジェクトのほうで組み立てるようにします。確保はほかのクラスやコンポーネントと同じく，

```
C++Builder    ScreenBitmap = new Graphics::TBitmap;
Delphi        ScreenBitmap := TBitmap.Create;
```

としてから，「Width」「Height」プロパティにフォームに貼られたTImageコンポーネントと同じ値を設定します。

このほかに，画面へ表示したい「線」「文字」といったものも，とにかくこのTBitmapオブジェクトにすべて描くようにします。つまり同サイズのTBitmapオブジェクトを仮想画面として利用するのです。

この描画作業が全部終わったら，Drawメソッドなどで TImageコンポーネントに，

```
C++Builder    Image1->Canvas->Draw(0, 0, ScreenBitmap);
Delphi        Image1.Canvas.Draw(0, 0, ScreenBitmap);
```

のように全部描画します(List 4-5，4-6)。これをゲームが終わるまで繰り返します(Fig. 4-3)。終わったら確保したTBitmapオブジェクトを，

```
C++Builder    delete ScreenBitmap;
Delphi        ScreenBitmap.Free;
```

として解放します。

ほかにも方法はいろいろあるのですが，試してみた結果この方法がもっともちらつきません。あとはDirectDrawなどを利用することになります。

List 4-5●画面のちらつきを抑える処理(Delphi)

```
ScreenBitmap: TBitmap;

procedure TForm1.FormCreate(Sender: TObject);
begin
    ScreenBitmap := TBitmap.Create;
    ScreenBitmap.Width = Image1.Width;
    ScreenBitmap.Height = Image1.Height;
end;

procedure TForm1.View;
begin
    Image1.Canvas.Draw(0, 0, ScreenBitmap);
    Update;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    ScreenBitmap.Free;
end;
```

List 4-6●画面のちらつきを抑える処理(C++Builder)

```
Graphics::TBitmap *ScreenBitmap;

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    ScreenBitmap = new Graphics::TBitmap;
    ScreenBitmap->Width = Image1->Width;
    ScreenBitmap->Height = Image1->Height;
}

void __fastcall TForm1::View(void)
{
    Image1->Canvas->Draw(0, 0, ScreenBitmap);
    Update();
}

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    delete ScreenBitmap;
}
```

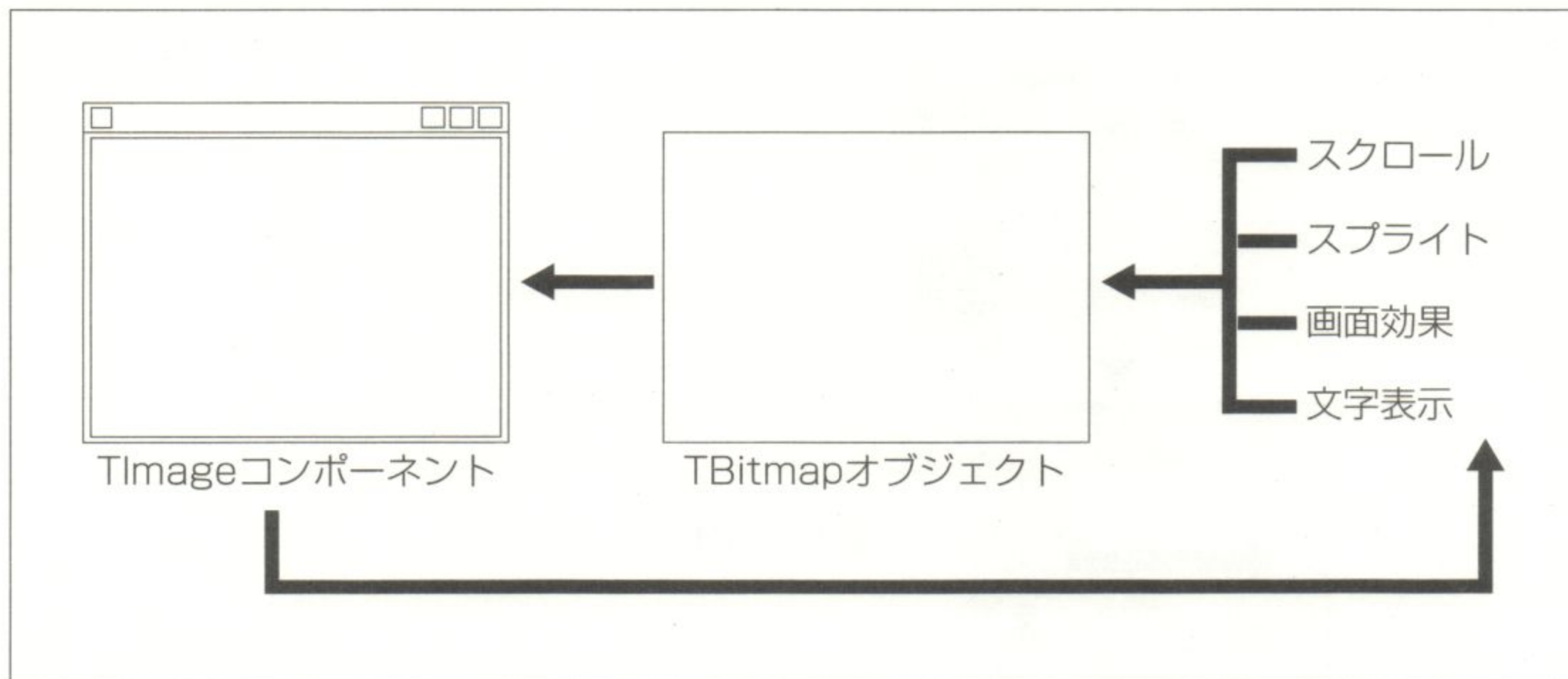



Fig. 4-3 ●フォームに貼り付けたTImageコンポーネントと同じサイズのTBitmapオブジェクトを仮想画面として用意し，画面表示に必要な処理はすべてTBitmapオブジェクト上で行い，処理が終了したらTImageコンポーネントに描画する

● 背景のスクロール処理

スクロールの基本は，「元の絵を指定した方向にずらす」ことです。スクロール全体の作業を見ると，Fig. 4-4のように「絵をずらす」，「ずらした部分に新しい絵を転送する」ことになります。ずらしたり絵を転送するときは，「そのようすをプレイヤーに見えなくする」「ちらつきを防ぐ」意味から仮想画面で行います。

◆表示部分をずらしてスクロールさせる

なお，表示している絵の「視点を限定できる」ものなら，この位置を動かすだけでよいこともあります(Fig. 4-5)。額縁よりも大きい絵があると考えてみてください。額縁の位置を動かすと絵の別の部分が見えてきます。たとえばTImageコンポーネントでは「コンポーネントそのもののWidth，Height」は額縁のように「データを表示する大きさ」，「Picture.BitmapのWidth，Height」は「絵そのものの大きさ」というように意味が違います。そこで「表示している部分」が絵本来の大きさよりも小さい場合は，Fig. 4-7のように「絵を表示している位置」だけを変えればスクロールしているように見せることができます。

ちょっとしたものならこれだけで十分ですが，背景全体の絵が大きくなってしまうロールプレイングゲームなどではこの方法は不向きです。

絵をずらす

空いた部分にデータを転送する

画面へ

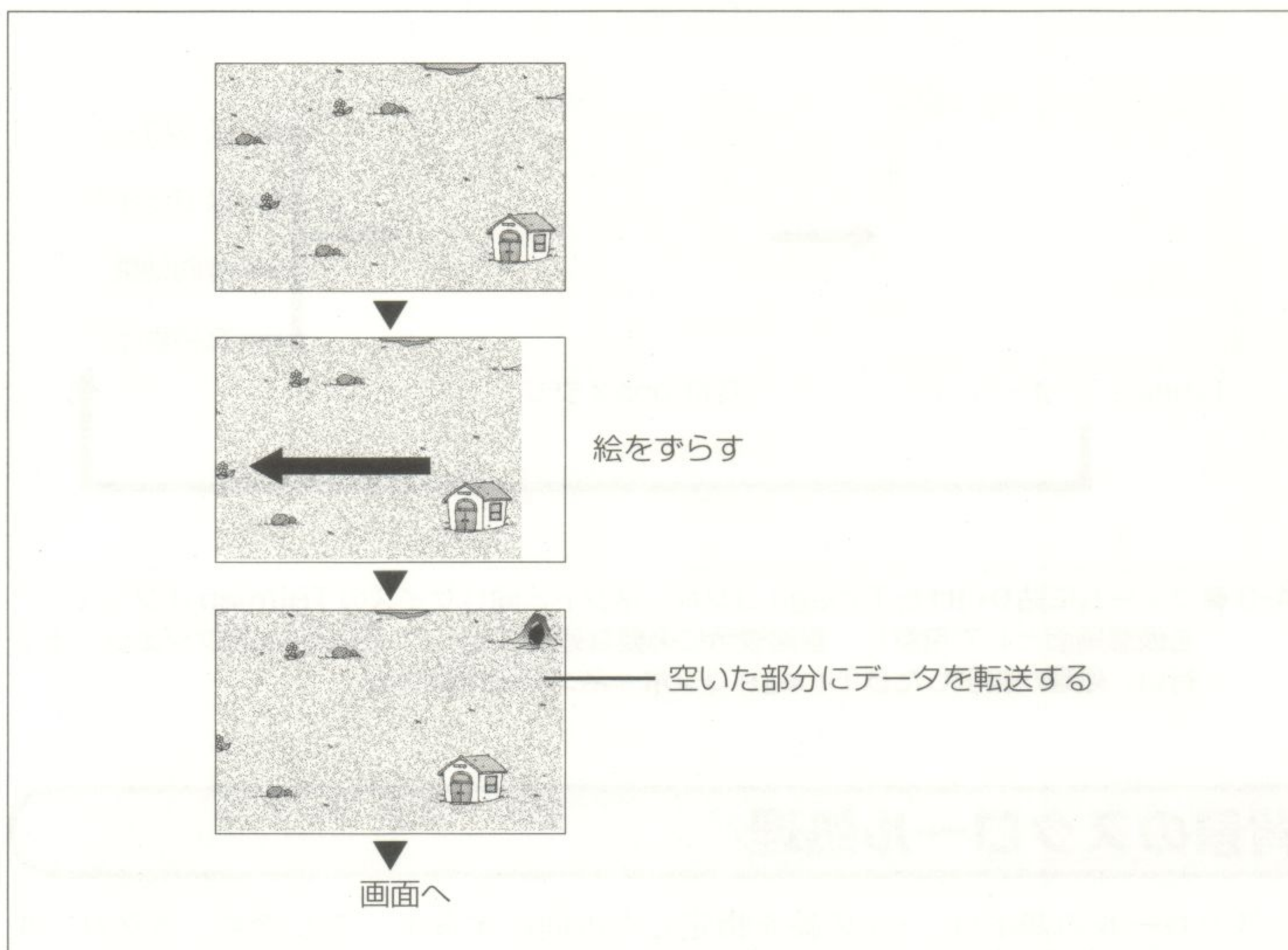



Fig. 4-4●スクロールの方法。絵をキャラクタの進行方向と逆方向にずらし，空いた部分に絵のデータを転送する

実際に画面上で見えている部分

背景

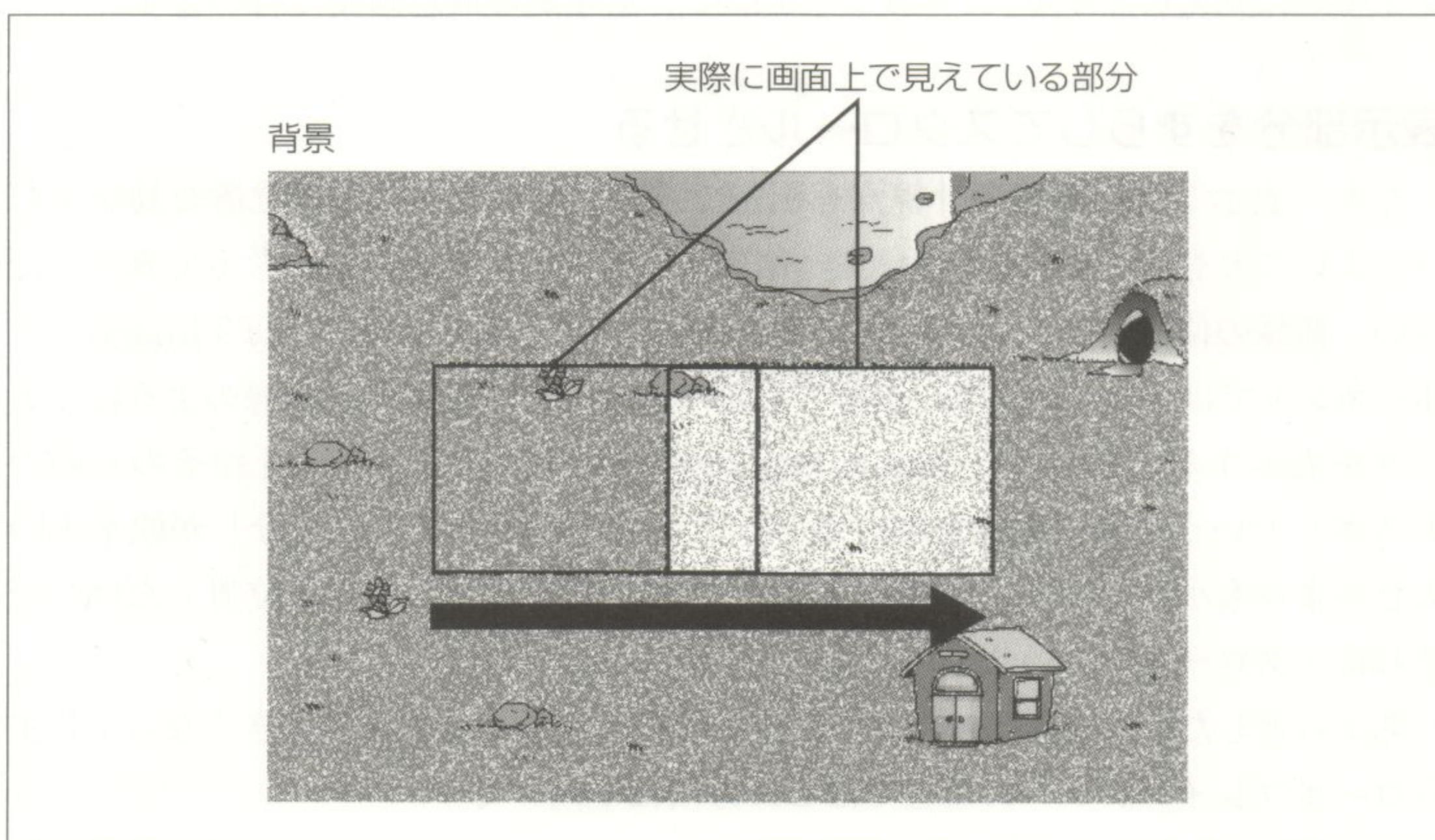



Fig. 4-5●1枚の背景の絵のうち，実際に画面上で見えている部分を変えることでスクロールしているように見せることができる

◆表示部分をずらす方法

表示されているグラフィックをずらすには，APIの利用とTCanvasオブジェクトの「CopyRect」メソッドを利用する方法の2つがあります。

APIのほうはメモリ中のビットマップデータを格納している仮想画面「Device Context(DC)」をずらす「ScrollDC」と，ウィンドウに表示されている絵をずらす「ScrollWindow」の2つがあります。これらは各コンポーネントの基本クラスとなっているTWinControlなどでも使用されているAPIです。

CopyRectメソッドは「指定した範囲のグラフィックデータをコピーする」だけのメソッドですが，「同じグラフィックデータをコピーする」とスクロールとしても利用できます。たとえば左側のデータを右側へ16ピクセルの差を付けてコピーすれば，それだけスクロールしたことになります(Fig. 4-9)。

これらを使ってスクロールを考えると「スクロール方向」と「スクロールの量」から「指定されたグラフィックデータをそれだけずらす」ことをして，そこに「ずれた分だけグラフィックデータをコピー」すればよいはずです。コンポーネントとして作るのなら，スクロールを実行するメソッドを用意して，ずれたデータをコピーする部分は「イベントハンドラ」として提供します。ゲームによって背景はパーツで構成されることもありますし，1枚の大きなグラフィックデータとなることもあるので，「グラフィックデータを組み立てる」部分は含まないようにします(List 4-7，4-8)。

List 4-7●スクロールの例(Delphi)

```
procedure TScroll.Scroll(Direction,
                         MoveCount: integer; SrcBitmap: TBitmap);
var
  SrcRect, DestRect, CopyDestRect: TRect;
begin
    if Direction and SCROLL_UP then begin
        SrcRect := Rect(0, 0, SrcBitmap.Width,
                                   SrcBitmap.Height - MoveCount);
        DestRect := Rect(0, MoveCount,
                                SrcBitmap.Width, SrcBitmap.Height);
        CopyDestRect := Rect(0, 0, 0, MoveCount);
        ScrollExec(SrcRect, DestRect, SrcBitmap);
        DoScrollCopy(CopyDestRect, SrcBitmap);
    end else if Direction and SCROLL_DOWN then begin
        SrcRect :=  Rect(0, MoveCount,
                                SrcBitmap.Width, SrcBitmap.Height);
        DestRect := Rect(0, 0, SrcBitmap.Width,
                                     SrcBitmap.Height - MoveCount);
        CopyDestRect := Rect(0, SrcBitmap.Height - MoveCount,
```

▼

List 4-7

```
                              SrcBitmap.Width, SrcBitmap.Height);
        ScrollExec(SrcRect, DestRect, SrcBitmap);
        DoScrollCopy(CopyDestRect, SrcBitmap);
    end else if Direction and SCROLL_LEFT then begin
        SrcRect := Rect(0, 0, SrcBitmap.Width - MoveCount,
                                                  SrcBitmap.Height);
        DestRect := Rect(MoveCount, 0, SrcBitmap.Width,
                                                 SrcBitmap.Height);
        CopyDestRect := Rect(0, 0, MoveCount, SrcBitmap.Height);
        ScrollExec(SrcRect, DestRect, SrcBitmap);
        DoScrollCopy(CopyDestRect, SrcBitmap);
    end else if Direction and SCROLL_RIGHT then begin
        SrcRect := Rect(MoveCount, 0, SrcBitmap.Width,
                                                 SrcBitmap.Height);
        DestRect := Rect(0, 0, SrcBitmap.Width - MoveCount,
                                                  SrcBitmap.Height);
        CopyDestRect := Rect(SrcBitmap.Width - MoveCount, 0,
                              SrcBitmap.Width, SrcBitmap.Height);
        ScrollExec(SrcRect, DestRect, SrcBitmap);
        DoScrollCopy(CopyDestRect, SrcBitmap);
    end;
end;

procedure TScroll.ScrollExec(SrcRect, DestRect: TRect;
                                             SrcBitmap: TBitmap);
begin
    SrcBitmap.Canvas.CopyRect(DestRect, SrcBitmap.Canvas, SrcRect);
end;

procedure TScroll.DoScrollCopy(DestRect: TRect; SrcBitmap: TBitmap);
begin
    if Assigned(FOnScrollCopy) then begin
        FOnScrollCopy(DestRect, SrcBitmap);
    end;
end;
```

List 4-8●スクロールの例(C++Builder)

```
void __fastcall TScroll::Scroll(int Direction, int MoveCount,
  Graphics::TBitmap *SrcBitmap)
{
   TRect SrcRect, DestRect, CopyDestRect;

    if (Direction & SCROLL_UP) {
        SrcRect = Rect(0, 0, SrcBitmap->Width,
                                SrcBitmap->Height - MoveCount);
        DestRect = Rect(0, MoveCount, SrcBitmap->Width,
                                        SrcBitmap->Height);
```

```
        CopyDestRect = Rect(0, 0, 0, MoveCount);
        ScrollExec(SrcRect, DestRect, SrcBitmap);
        DoScrollCopy(CopyDestRect, SrcBitmap);
    } else if (Direction & SCROLL_DOWN) {
        SrcRect =  Rect(0, MoveCount, SrcBitmap->Width,
                                          SrcBitmap->Height);
        DestRect = Rect(0, 0, SrcBitmap->Width,
                                  SrcBitmap->Height - MoveCount);
        CopyDestRect = Rect(0, SrcBitmap->Height - MoveCount,
                            SrcBitmap->Width, SrcBitmap->Height);
        ScrollExec(SrcRect, DestRect, SrcBitmap);
        DoScrollCopy(CopyDestRect, SrcBitmap);
    } else if (Direction & SCROLL_LEFT) {
        SrcRect = Rect(0, 0, SrcBitmap->Width - MoveCount,
                                          SrcBitmap->Height);
        DestRect = Rect(MoveCount, 0, SrcBitmap->Width,
                                          SrcBitmap->Height);
        CopyDestRect = Rect(0, 0, MoveCount, SrcBitmap->Height);
        ScrollExec(SrcRect, DestRect, SrcBitmap);
        DoScrollCopy(CopyDestRect, SrcBitmap);
    } else if (Direction & SCROLL_RIGHT) {
        SrcRect = Rect(MoveCount, 0, SrcBitmap->Width,
                                          SrcBitmap->Height);
        DestRect = Rect(0, 0, SrcBitmap->Width - MoveCount,
                                          SrcBitmap->Height);
        CopyDestRect = Rect(SrcBitmap->Width - MoveCount, 0,
                            SrcBitmap->Width, SrcBitmap->Height);
        ScrollExec(SrcRect, DestRect, SrcBitmap);
        DoScrollCopy(CopyDestRect, SrcBitmap);
    }
}

void __fastcall TScroll::ScrollExec(TRect SrcRect, TRect DestRect,
  Graphics::TBitmap *SrcBitmap)
{
    SrcBitmap->Canvas->CopyRect(DestRect, SrcBitmap->Canvas, SrcRect);
}

void __fastcall TScroll::DoScrollCopy(TRect DestRect,
  Graphics::TBitmap *SrcBitmap)
{
    if (FOnScrollCopy) {
        FOnScrollCopy(DestRect, SrcBitmap);
    }
}
```

Section

3 TImageListを利用した擬似スプライト

高度なグラフィック処理を必要とするゲームプログラムでは，スプライトは欠かすことのできない機能といえます。そこでC++BuilderとDelphiを使って擬似的にスプライトを作ってみます。

TImageListコンポーネントの利用方法

スプライトには「表示するパーツの座標管理」「グラフィックの合成」の2つの機能があります。C++BuilderとDelphiでは，画像の合成方法として「TBitmapのTransparent機能を利用する」，「マスクと論理演算を使った画像コピーを行う」，「TBitmapのScanLineプロパティを使って透過させないピクセルだけを単純にコピー」，「TImageListコンポーネント」など多くの選択肢があります。ここではTImageListコンポーネントを使ってみることにします。

TImageListコンポーネントは，ビットマップデータやアイコンデータを「イメージリスト」という「配列」に管理して，指定されたところに「合成表示」することができます。このコンポーネントを使う際には「Width」「Height」プロパティへ「ビットマップデータの大きさ」を指定しなければなりません。合成するデータはすべてこの大きさになります。

◆AddMaskedメソッドでマスクデータを作る

合成するビットマップデータには，そのままでは合成処理に必要なマスクデータがありません。そこでマスクデータを作るために「AddMasked」メソッドを利用します。AddMaskedメソッドを使ってビットマップデータをイメージリストに追加すると，マスクデータが作られます。このとき2番目の引数に，背景にしたい部分を示す「透過色」を指定します。そのためビットマップデータは，背景となる部分を単一の色とします。また，あたりまえなのですが透かしたくない部分には透過色を使えません。これに気をつけないと思わぬ絵が表示されてしまいます。

◆Drawメソッドで合成する

イメージリストに収めたビットマップデータを背景などと合成するには，「Draw」

メソッドを使います。「X，Y」はイメージリストのビットマップデータを背景に表示する座標，「Index」は表示したいグラフィックデータが含まれているイメージリストの位置を指定します。AddやAddMaskedすると，「そのビットマップデータが格納されたイメージリストの位置」の値が返ってきます。描画するときはこの値が必要になります。

Indexプロパティに「Count」プロパティの値を－1したものを指定するとイメージリストのいちばん最後，つまりいちばん最近に設定されたビットマップデータが描画されます。これを応用して，0からこの数だけループしながらIndexの値を増加させてDrawメソッドを実行すると，イメージリストに含まれている全部のビットマップデータを描画することができます。

● 擬似スプライトの機能

TImageListコンポーネントを利用した擬似スプライトの機能を考えてみます。ゲーム本体から疑似スプライトを使うとすると，始めは「スプライトレジスタへの画像や絵の設定」，ゲーム中では「背景とスプライトの合成」の2つの機能が必要になります(Fig. 4-6)。

これらをまとめて1つのコンポーネントにしてしまいます。

ゲーム本体		スプライト
パーツデータをスプライトレジスタに登録	▶	スプライトレジスタ
▼		
登録した内容をイメージリストに設定	▶	イメージリスト設定
ゲーム中 ▼		
背景とスプライトのグラフィックデータを合成	▶	合成
▼		
座標の変更	▶	スプライトレジスタの変更

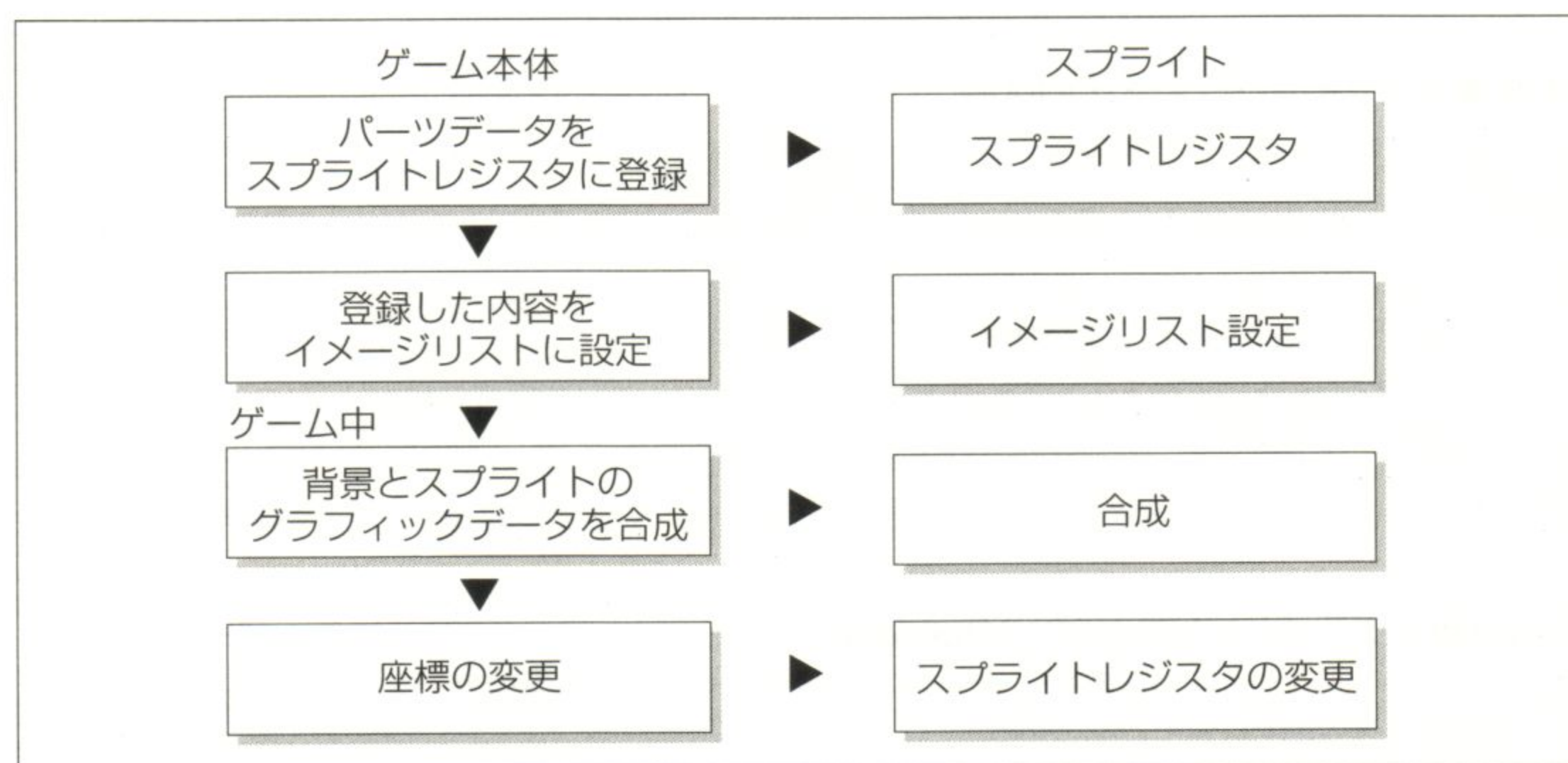



Fig. 4-6 ● TImageListコンポーネントを使った疑似スプライトの機能

◆スプライトレジスタへ値を収める

スプライトにつきものなのが，座標などを収めるための「スプライトレジスタ」

です。まずはこれを構造体として定義します(List 4-9，4-10)。

構造体が定義できたら，次はスプライトレジスタに必要な値を収めていきます。スプライトレジスタに収める内容は「表示位置のX，Y座標」「ビットマップデータへのポインタ」などです。さらに「このスプライトレジスタが空いているかどうか」「このスプライトレジスタを使うかどうか」の各フラグを含めます。このフラグはビットマップデータの設定を行うときなどに必要になります。「イメージリストの番号を格納する変数」も加えておきます。

スプライトには「重ねる順番」も必要です。ゲームによってはキャラクタが木などの陰に隠れてしまうことがありますが，これを実現するには「キャラクタを収めているスプライトレジスタよりも，木を収めたスプライトレジスタを先に合成する」ことを指定しなければなりません。この「優先順位」を示す変数もスプライトレジスタには必要です。

スプライト機能では直接使わない変数「Tag」プロパティもスプライトレジスタへ入れておくと，あとで何かと便利です。ゲーム中では，パーツに対して「動く方向を示したスクリプト」など固有の値が必要となることもあります。そんなときは，このTagプロパティがあればそれを利用するだけで済むようになります。TImageListコンポーネントにもこれと同じものが付いています。

完成したスプライトレジスタの構造体は，配列(テーブル)の形にしておきます。

List 4-9 ●スプライトレジスタ(Delphi)

```
type
  TSpReg = record
    PartsBitmap: TBitmap;
    ListIndex, x, y: Integer;
    FlagEmpty, Enabled: Boolean;
    Tag: Integer;
  end;
```

List 4-10 ●スプライトレジスタ(C++Builder)

```
typedef struct {
    Graphics::TBitmap *PartsBitmap;
    int ListIndex, x, y;
    bool FlagEmpty, Enabled;
    int Tag;
} TSpReg;
```

◆ビットマップデータの設定

ビットマップデータの設定には，ちょっと難関があります。前述のようにTImage Listコンポーネントは，「設定するすべてのビットマップが同じ大きさ」でなければいけません。しかし実際のゲームプログラムでは，大きさがまちまちのパーツを利用しなければならないこともあります。これを解決するには「いちばん大きなパーツに合わせて，ほかのパーツの大きさを揃える」ことがもっとも簡単な方法です。そこで最初は，「スプライトに使うビットマップデータのポインタをスプライトレジスタに設定」してやり，全部の設定が終わったら「ビットマップデータの大きさを揃える」ことをします。これはべつべつの関数で提供します。

ビットマップデータの大きさを揃えるには，始めに「スプライトレジスタの数だけビットマップデータの大きさを調べて，もっとも大きいサイズを得る」ことをして，それから「TImageListコンポーネントに大きさを設定」「テンポラリ用ビットマップへコピー」「AddMaskedで設定」という流れになります(List 4-11，4-12)。作業が多くてめんどうに思えますが，TImageListコンポーネントを使うためには仕方のないところです。

List 4-11 ●イメージリストへの設定(Delphi)

```
procedure TSp.MakeList;
var
  Width, Height, i: Integer;
  TmpBitmap: TBitmap;
begin
    TmpBitmap := TBitmap.Create;
    Width := 0;
    Height := 0;
    { 大きさを出す }
    for i := 0 to REGTBLSIZE-1 do begin
        if not SpRegTbl[i].FlagEmpty then begin
            if SpRegTbl[i].PartsBitmap.Width > Width then
                Width := SpRegTbl[i].PartsBitmap.Width;
            if SpRegTbl[i].PartsBitmap.Height > Height then
                Height := SpRegTbl[i].PartsBitmap.Height;
        end;
    end;
    { 大きさの設定 }
    ImageList.Width := Width;
    ImageList.Height := Height;
    { ビットマップデータの設定 }
    for i := 0 to REGTBLSIZE-1 do begin
        if (not SpRegTbl[i].FlagEmpty) and (SpRegTbl[i].Enabled) then
begin
```

▼

List 4-11

```
                SpRegTbl[i].ListIndex := ImageList.Count;
                TmpBitmap.Assign(SpRegTbl[i].PartsBitmap);
                TmpBitmap.Width := ImageList.Width;
                TmpBitmap.Height := ImageList.Height;
                FillBitmap(TmpBitmap, FBkColor);
                TmpBitmap.Canvas.Draw(0, 0, SpRegTbl[i].PartsBitmap);
                ImageList.AddMasked(TmpBitmap, FBkColor);
            end;
        end;
        TmpBitmap.Free;
    end;
```

List 4-12●イメージリストへの設定（C++Builder）

```
void __fastcall TSp::MakeList(void)
{
    int Width = 0, Height = 0;
    Graphics::TBitmap *TmpBitmap = new Graphics::TBitmap;

    // 大きさを出す
    for (int i = 0; i < REGTBLSIZE; i++) {
        if (!SpRegTbl[i].FlagEmpty) {
            if (SpRegTbl[i].PartsBitmap->Width > Width)
                Width = SpRegTbl[i].PartsBitmap->Width;
            if (SpRegTbl[i].PartsBitmap->Height > Height)
                Height = SpRegTbl[i].PartsBitmap->Height;
        }
    }
    // 大きさの設定
    ImageList->Width = Width;
    ImageList->Height = Height;
    // ビットマップデータの設定
    for (int i = 0; i < REGTBLSIZE; i++) {
        if ((!SpRegTbl[i].FlagEmpty) && (SpRegTbl[i].Enabled)) {
            SpRegTbl[i].ListIndex = ImageList->Count;
            TmpBitmap->Assign(SpRegTbl[i].PartsBitmap);
            TmpBitmap->Width = ImageList->Width;
            TmpBitmap->Height = ImageList->Height;
            FillBitmap(TmpBitmap, FBkColor);
            TmpBitmap->Canvas->Draw(0, 0, SpRegTbl[i].PartsBitmap);
            ImageList->AddMasked(TmpBitmap, FBkColor);
        }
    }
    delete TmpBitmap;
}
```

このような仕組みだと，いったんTImageListコンポーネントにビットマップデータを設定したあとに，スプライトレジスタのビットマップデータを取り換えよう

とすると，ややめんどうです。そこで「ビットマップデータを変更する場合は，始めに設定したビットマップデータよりも大きくしない」ことにします。ゲーム中にビットマップデータの変更が必要になるものは，「体力ゲージ」「時間表示」パーツのアニメーションなどがほとんどで，大きさはほとんど変わらないものばかりなはずです。パーツをアニメーションさせるときには，全部のパーツをTImageListコンポーネントに設定してから，スプライトレジスタの「FlagEnpty」プロパティを全部Trueにします。その設定した番号をスプライトレジスタの「ListIndex」プロパティへ入れるようにしておけば，絵が切り換わるようになります。これで問題が起きるようならもう1つビットマップの大きさの違うスプライトコンポーネントのオブジェクトを作って，そこで処理するようにします。

◆疑似スプライトでの合成処理

背景との合成は，「背景を収めたTBitmapデータ」「スプライトレジスタの座標とIndex値」を基に TImageListコンポーネントの「Drawメソッドで描画する」ことをスプライトレジスタの数だけ繰り返します。このときにスプライトレジスタの「空き」「使用可能かどうか」のフラグを見て，「使用中」「使用可能」のレジスタだけ合成するようにします(List 4-13，4-14)。これで完成です。

List 4-13●スプライトの合成(Delphi)

```
procedure TSp.View(SrcBitmap: TBitmap);
var
  i: integer;
begin
    for i := 0 to REGTBLSIZE-1 do begin
        if (not SpRegTbl[i].FlagEmpty) and (SpRegTbl[i].Enabled) then
begin
            ImageList.Draw(SrcBitmap.Canvas,
              SpRegTbl[i].x, SpRegTbl[i].y, SpRegTbl[i].ListIndex);
        end;
    end;
end;
```

List 4-14●スプライトの合成(C++Builder)

```
void __fastcall TSp::View(Graphics::TBitmap *SrcBitmap)
{
    for (int i = 0; i < REGTBLSIZE; i++) {
        if ((!SpRegTbl[i].FlagEmpty) && (SpRegTbl[i].Enabled)) {
            ImageList->Draw(SrcBitmap->Canvas,
```

▼

List 4-14

```
                SpRegTbl[i].x, SpRegTbl[i].y, SpRegTbl[i].ListIndex);
            }
        }
    }
```

● スプライトの応用

スプライトというものは，ただ単に「背景へパーツを合成する」以外にもさまざまな応用ができます。背景そのものもいくつかパーツを組み合わせて作ると，背景に奥ゆきが出てきます。たとえばビルの屋上から下を見下ろしているという背景にしたいとします。いちばん下になる部分は道路や流れる車のようすです。これはアニメーションパターンとして持たせます。林立する近くのビルはパーツとして作り，道路部分と合成します。見下ろしているビルも奥にいくほど縦が狭くなるパーツに分けておきます。

もしこの状態で横方向に動いたとき，パーツとなっているビルは見えている面などが少しずつ動きます。実際にビルから下をのぞいて，そのままの状態で横に動いてみるとわかると思います。いちばん手前の斜面となっているビルの面は，奥にいくほど狭い範囲でスクロールしているように見えます(Fig. 4-7)。このように遠近法に従って手前と奥のスクロール量を変えてスクロールさせることを「ラスタスクロール」と呼んでいます。このようにスプライトはアイデア次第でいろいろなものに応用できるので，ぜひ使ってみてください。

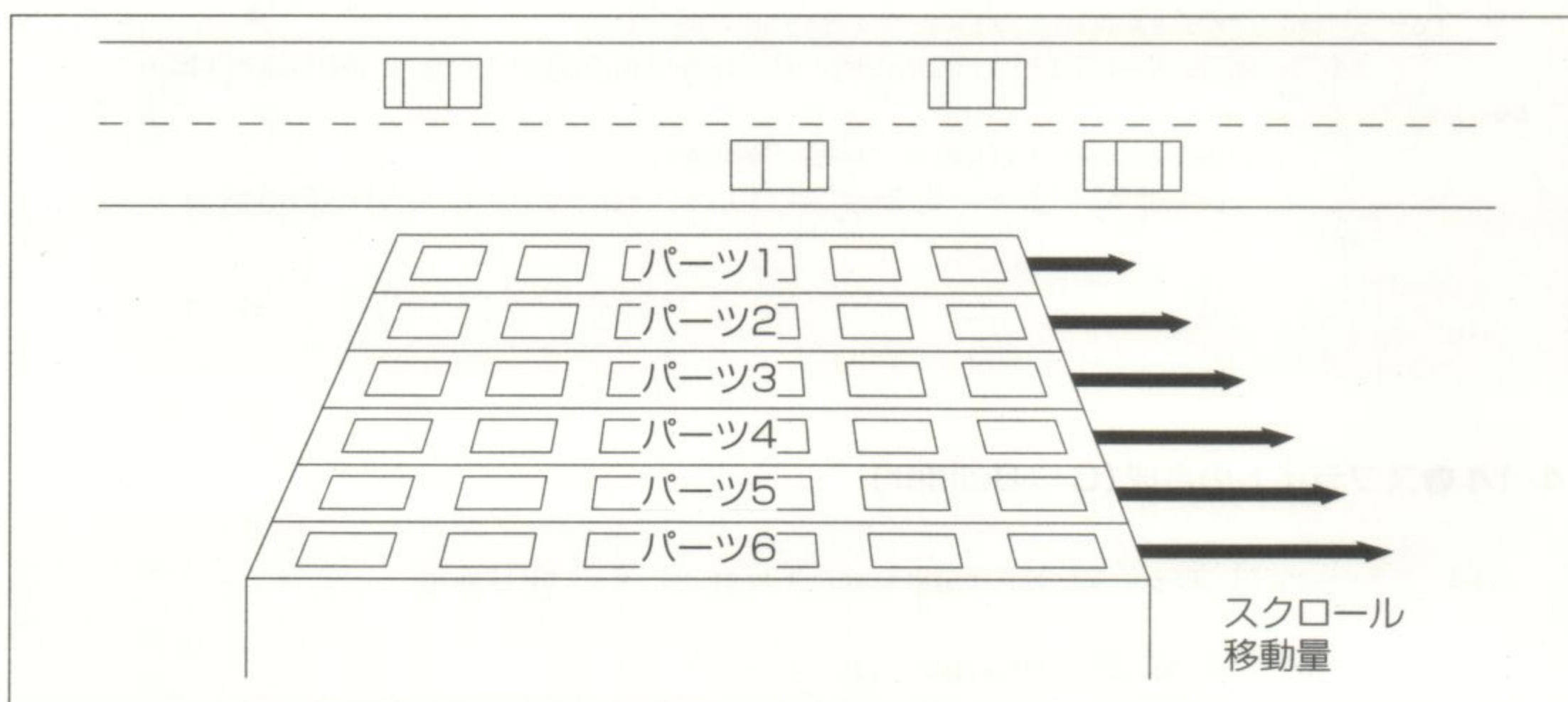



Fig. 4-7 ●スタスクロールの例。遠近法に従って配置されたパーツを，奥にいくに従って移動量を小さくしてスクロールさせる

あとがき

連載中はさまざまなゲームをかたっぱしから作っていった気がします。ここでは実際のゲーム制作のようすを紹介しながら，順を追ってその過程を解説して，あとがきにかえたいと思います。

●まずは企画書を書く

すべてのゲーム制作で始めに行ったことは企画書作りです。付属CD-ROMにはサンプルプログラムを作るのに使ったすべての企画書が収められています。キャラクタやゲームの設定，ゲーム画面の構成などを，私やグラフィック担当のMALUMI氏がお互いに出しあったアイデアを基にして書いたものです。市販されるゲームの企画書では，ゲームの売り口上やスタッフの配置などといったことも書き加えられるのですが，ここでは簡単に必要なものだけを書いています。

この企画書からキャラクタデザインやゲーム制作が進められます。とはいっても，これを完全に実現させるわけではありません。実際にサンプルプログラムと企画書を比べればわかるとおり，ゲーム完成までに途中で削られたり変更されたりする機能もあります。なぜ文章にするかというと，頭の中で考えているよりももっと明確で具体的なイメージを形作ることができ，それに対して他人のアドバイスを求めることができるからです。ストーリ主体のゲームならこうした作業を行ったほうがなおさら能率的だと思います。またグラフィックをMALUMI氏に依頼するうえで，「どういったイメージになるのか」を伝えてゲームの雰囲気に合った絵を描いてもらうために「明文化して間違いを防ぐ」といった意味もあります。

●必要なツールを作る

次は「ゲームを作るためのツール」を作ります。ロールプレイングゲームでは，マップを作成するのに使う「マップエディタ」を作成しました。マップ表示プログラムを拡張して「パーツの取り換え」「マップのセーブ/ロード」「各フラグの処理」「全体図の表示」などを追加します。

これ以外にスプライト機能の前身として，パーツを透過して重ねたりするツールも作りました。こうしたツールたちを作ることによって，実はゲームシステムの表示に関することがいつのまにかできあがったことになります。

●シナリオを書く

だいたいここまでは原稿執筆と平行作業です。ある程度原稿ができてくると，MALUMI氏からキャラクタデザインのラフやパーツが送られてきます。これに合わせて燃えてきたところでラフなシナリオを書きあげます。ゲームのシナリオは普通の小説とは違い，「ゲームのシステムの中で読ませる」ことを考えなくてはいけません。1つのシーンで同じキャラクタがえんえんと喋っているようではプレイヤーは飽きてしまいます。逆に短い文章ばかりがずっと続いても困ります。シーンごとに文量のバランスを取ることが必要です。それを逆手に取ってシナリオの効果にすることもあります。

●ゲーム本体のプログラミング

燃えるシナリオやグラフィックデータが揃ってきたら次にいよいよゲーム本体のプログラミングです。原稿や企画書に書いた動作を，DelphiやC++Builderで実現させていきます。スプライトなどいくつかの道具がすでに揃っていれば，それを利用するようにして作ります。あとは時間との闘いです。このときに，作るのに時間がかかりそうな処理はどんどん省かれていきます。

こうしてサンプルプログラムの完成です。こう書くといかにもすらすらとできてきたように思いますが，この間にはさまざまな紆余曲折がありました。連載3回目ぐらいまでは余裕があったのですが，それ以降はいつも時間との闘いとなってしまいました。とくにフラグの塊であったロールプレイングゲームと，仕組みが複雑になってしまった麻雀ゲームの2つはいまでもその苦労を覚えています。その逆にカードゲームは実質3日ほどで完成しています。

このようにゲームの種類によってさまざまですが，ほぼ月1回の割合でそれなりのゲームを作っていきました。やってみればできるものです(笑)。

付属CD-ROMの使い方

付属CD-ROMには，本書に掲載されたすべてのリスト，各サンプルゲーム，サンプルゲームのソース，サンプルゲームに必要なDirectX 8.0a SDKが収録されています。

◆本書掲載リスト

＜¥list＞フォルダには，本書に掲載されたすべてのリストが収録されています。

◆コンポーネント

＜¥class＞フォルダには，本書で作成したDelphi/C++Builder用のコンポーネントが収録されています(Table 1)。DelphiとC++Builder特有のクラスや関数を使用しているので，そのままではほかの処理系でコンパイルすることはできません。

◆サンプルゲームのソース

＜¥sample¥source¥delphi＞フォルダと＜¥sample¥source¥c_build＞フォルダには，それぞれDelphi，C++Builder用のサンプルゲームのソースファイルが，各ゲームごとのフォルダに分けて収録されています(Table 2)。

◆ WonderWitch

＜¥WonderWitch＞フォルダには，『C MAGAZINE』誌(ソフトバンクパブリッシング)の2000年12月号～2001年5月号に掲載された「遊びのレシピfor Wonder Witch」の掲載ファイルと誌面をPDFファイルにしたものが収録されています。

◆ DirectX 8.0a SDK

＜¥DirectX＞フォルダには，格闘ゲームのプレイ/コンパイルに必要なDirectX 8.0a SDKが，マイクロソフト(株)のご協力により収録されています。DirectX 8.0a SDKは，DirectXを使用したアプリケーションの開発を行うための開発キットです。

インストールには，＜¥DirectX＞フォルダにあるinstall.exeを実行してください。DirectX 8.0a SDKのご使用方法などについては，各フォルダ内のテキストファイルをご覧ください。

CD-ROMに含まれる「Microsoft DirectX 8.0a Software Development Kit(日本

語ヘルプ付)」(以下，DirectX 8.0a SDK)の権利は，Microsoft Corporationに帰属し，その使用およびこれに基づく製品の開発，頒布は，DirectX 8.0a SDKのセットアップ時に表示される使用許諾書(英語)，およびDirectX 8.0a SDKを含むディレクトリ内のDXF¥LICENSE¥REDIST.TXT(英文)の各条件に従わなければなりません。DirectX 8.0a SDKの使用または使用不能から生じるいかなる損害についてもMicrosoft Corporationは一切責任を負いません。DirectX 8.0a SDKの内容に関するお問い合わせ，サポートは受け付けておりません。DirectX 8.0aは「NEC PC-9800/9821シリーズ」はサポートしていません。日本語ヘルプはCDの¥DirectX¥dxf¥doc¥DX8Jhelpにあります。インストーラからはインストールされませんのでコピーしてご利用ください。

Table 1●ゲーム用のコンポーネント

コンポーネント	ファイル名	内容
ジョイスティック	joy.cpp joy.h joy.pas	ジョイスティックからの入力を制御する
シナリオコンポーネント	scn.cpp scn.h scn.pas	シナリオを制御する
スプライトコンポーネント	sp.cpp sp.h sp.pas	スプライト制御

Table 2●サンプルゲームとそのソースの収録フォルダ

フォルダ名	ゲーム	掲載
＜¥adv＞	アドベンチャゲーム	Chapter2，Section1
＜¥rpg＞	ロールプレイングゲーム	Chapter2，Section2
＜¥maze＞	ダンジョンゲーム	Chapter2，Section3
＜¥sim＞	シミュレーションゲーム	Chapter2，Section4
＜¥shoot＞	シューティングゲーム	Chapter2，Section5
＜¥card＞	カードゲーム	Chapter2，Section6
＜¥mahjong＞	麻雀	Chapter2，Section7
＜¥fight_old＞	格闘ゲーム	Chapter2，Section8

改訂第2版
ゲームプログラミング 遊びのレシピ
アルゴリズムとデータ構造

2001年6月15日　初版1刷発行

著　者：有馬　元嗣（ありま　もとつぐ）
発行者：稲葉　俊夫
発行所：ソフトバンク パブリッシング株式会社
〒107-8790　東京都港区赤坂4-13-13
販売　03-5549-1200
編集　03-5549-1143
サポートURL　http://cmaga.zdnet.co.jp/books/recipe/

制　作：株式会社アレフ　　本文イラスト　：工藤　研一
印　刷：文唱堂印刷株式会社
装　丁：花本　浩一

本書の内容に関するご質問は，必ず書面かFAX(03-5549-1147)にて，「C MAGAZINE編集部」までお願いします。電話でのお問い合わせには応じかねますので，ご了承ください。
乱丁・落丁は小社販売局にてお取り替えいたします。
定価はカバーに記載されています。

Printed in Japan　　ISBN4-7973-1653-5

SOFT BANK Publishing ソフトバンク パブリッシング

改訂第2版

ゲームプログラミング

遊びのレシピ

アルゴリズムとデータ構造

改訂第2版 ゲームプログラミング 遊びのレシピ アルゴリズムとデータ構造 有馬元嗣／著 SOFT BANK Publishing

ゲームプログラミング

遊びのレシピ

アルゴリズムとデータ構造

CMBK-00002 HK5

改訂第2版

COMPACT disc

SOFT BANK Publishing

CMBK-00002
Made in Japan

CD-ROMの内容，使い方については，本文ならびにCD-ROMのreadme.txtをお読みください。